Contract-based Web Service Composition

DISSERTATION

zur Erlangung des akademischen Grades
doctor ingenieur
(Dr. Ing.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Ing. Nikola Milanovic
geboren am 22.03.1976 in Sarajevo

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Uwe Küchler

Gutachter:

1. Prof. Dr. Miroslaw Malek
2. Prof. Dr. Alexander Reinefeld
3. Prof. Dr. Roberto Baldoni

eingereicht am:	07. März 2006
Tag der mündlichen Prüfung:	13. Jun 2006

Abstract

Service-oriented architecture (SOA) is focused on building loosely coupled distributed systems with minimal shared understanding among system components. The main building blocks in SOA are services. Services are self-descriptive, self-contained, platform-independent and openly-available components that interact over the network. The main goal of SOA is transparent, flexible and dynamic interaction of services and their clients over multiple interconnected domains. While native capabilities of service-oriented architectures, such as description, discovery, communication and binding, have been well understood and standardized, the issue of service composition has not yet been satisfactorily solved.

This dissertation challenges the SOA postulate that service should disclose only basic functional signature, and demonstrates that based on semantic service description, including functional and non-functional properties, a viable solution for service composition can be developed (composable service architecture), that supports: 1) Extended descriptive and search capabilities by developing contract-based description language including non-functional properties such as security, dependability, timeliness; 2) Verification of composition correctness by modeling services as abstract machines and developing a formal composition language, and 3) Automatic service composition by treating automated and dynamic selection of composition partners as a search problem and developing search algorithms for that purpose. Finally, in order to show the viability of the proposed architectural solution, a prototype of Web Services composition server is described including design and implementation.

Keywords:
Web services, composition, verification, automatic composition

Zusammenfassung

Dienstorientierte Architekturen (SOA = Service Oriented Architecture) dienen dem Aufbau von lose miteinander verbundenen, verteilten Systemen, deren Komponenten eine minimale gemeinsame Systemsicht haben. Die wichtigsten Bausteine der SOA sind Dienste. Dienste sind selbstbeschreibende, eigenständige, plattform-unabhängige und frei verfügbare Komponenten, die über das Netzwerk interagieren. Das Hauptziel der SOA ist die transparente, flexible und dynamische Interaktion von Diensten und deren Benutzern innerhalb mehrerer zusammenhängender Domänen. Während die nativen Fähigkeiten von dienstorientierten Architekturen, wie Beschreibung, Entdeckung, Kommunikation und Bindung, bereits gut erfasst und standardisiert worden sind, ist das Problem der Dienstkomposition bisher noch nicht zufriedenstellend gelöst worden.

Diese Dissertation hinterfragt die Grundvoraussetzung der SOA, die darin besteht, dass lediglich die Grundfunktionsweise von Diensten offen gelegt werden sollte, und demonstriert, dass auf der Grundlage einer semantischen Dienstbeschreibung (einschließlich funktionaler und nichtfunktionaler Eigenschaften) eine praktikable Lösung zur Dienstkomposition entwickelt werden kann (komponierbare Dienstarchitektur). Diese Lösung erfordert 1) erweiterte deskriptive Fähigkeiten und Suchmöglichkeiten durch die Entwicklung einer vertragsbasierten Beschreibungssprache einschließlich nichtfunktionaler Eigenschaften wie Sicherheit, Verlässlichkeit und Rechtzeitigkeit; 2) das Feststellen der Kompositionskorrektheit durch die Modellierung von Diensten als abstrakte Maschinen und die Entwicklung einer formalen Kompositionssprache und 3) automatische Dienstkomposition, indem Suchalgorithmen für die automatisierte und dynamische Selektion von Kompositionspartnern entwickelt werden. Abschließend wird der Prototyp eines Kompositionsservers für Web Services einschließlich Design und Implementierung beschrieben, um die Realisierbarkeit der vorgeschlagenen Architektur aufzuzeigen.

Schlagwörter:
Web services, Komposition, Verifikation, automatische Komposition

Contents

List of Figures

Chapter 1

Introduction

1.1 Service-Oriented Architectures (SOA)

The term service-oriented architecture (SOA) emerged in [27] to describe the approach of building loosely coupled distributed systems with minimal shared understanding among system components. The main building blocks in SOA are services. Services are self-describing, open components that support rapid, low-cost development and deployment of distributed applications. The main goal of SOA is transparent, flexible and dynamic interaction of services and their clients over multiple interconnected domains. The benefits of SOA include increased efficiency through task outsourcing and component reuse, easier integration, increased flexibility and agility at business and IT level, development of composite applications, enabling of multi-vendor application sourcing, and on-demand interconnection with business partners. SOA can be deployed at different levels of granularity: from exposing fine-grained technical functions to coarse-grained business or scientific operations and processes.

At the time when the term SOA was coined, there were several existing architectures aspiring to become SOA standards, including HP e-speak [74], Sun JINI [104] and Web Services. Here is how the latter's specification defines a SOA as a distributed system in which agents, also known as services, coordinate by sending messages [173]:

> A Service-Oriented Architecture (SOA) is a form of distributed systems architecture that is typically characterized by the following properties:
>
> - Logical view: The service is an abstracted, logical view of actual programs, databases, business processes, etc., defined

in terms of what it does, typically carrying out a business-level operation.

- Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.
- Description orientation: A service is described by machine processable metadata. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.
- Granularity: Services tend to use a small number of operations with relatively large and complex messages.
- Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.
- Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

At the time that SOA emerged, enterprise computing was (and to some extent still is) dominated by component frameworks and monolithic n-tier applications. Compared with the newly emerging paradigm, component-based application servers offered mechanisms supporting dependability, security, transactions and other similar properties. The main problems that those applications faced, however, were complexity, maintenance and interoperability. SOA promised to solve these problems with the lightweight infrastructure offering clients agile and versatile collaboration with other organizations by

exposing their businesses or departments as Web Services (e.g., order procurement, finance, accounting, human resources, supply chains or manufacturing).

SOA is based on a model of roles. Every service can assume one (or more) roles in the SOA. Service providers offer services and publish the availability and description of their services. Service brokers (directories) register and categorize published services, and themselves provide a search service. Service requesters use broker services to find adequate services and invoke them. Such an architecture indeed provides advantages compared to monolithic n-tier applications, namely, complexity is reduced by eliminating service implementation issues, maintenance of a system is easy since different services can be plugged in and out of an application with relative ease, while interoperability is guaranteed by relying on standard communication protocols and simple broker request architecture (passing of text messages).

However, SOA is it still mainly used *inside* enterprises as a bridge and integrator of existing different systems that need to exchange data. Why do we rarely see true SOA developed applications that interconnect different enterprises in genuine business-to-business (B2B) fashion? To be able to answer this question, we have to gain a deeper understanding of Web Services, the most prominent SOA that exists today, as well as to explore the role that SOA plays in the broader scope of distributed systems.

1.2 Web Services

There are two accepted definitions of Web Services:

> A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards[1].

and

> A Web Service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems.

[1] http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/

> These systems may then interact with the Web Service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols.[2]

It can be seen that Web Services represent a SOA that relies on three key technologies:

- Web Service Description Language (WSDL) for description of Web Services and their relevant properties [32].
- Simple Object Access Protocol (SOAP) that is used for passing messages and invoking operations that Web Services offer [61].
- Universal Description, Discovery and Integration (UDDI) protocol that is used for creating Web Service directories and searching for adequate services [33].

Web Services architecture stack can be represented as shown in Figure 1.1. It comprises three main layers: basic services, composite services and managed services [120]. The first layer defines native capabilities, such as publication, discovery, selection and binding, that are realized with WSDL, SOAP and UDDI in the Web Services architecture. The second layer deals with the problem of service cooperation, defining coordination, conformance, monitoring and quality of service as key requirements. Finally, the third layer provides service level agreements mechanisms, certification and reputation systems, as well as operation assurance and support, since they are all needed in the economic analysis of B2B Web Services interactions.

The layer of Web Services native capabilities is well defined and standardized. WSDL is an XML description language that completely decouples service implementation and specification. Only the most basic functional description of a service is provided, namely the service endpoint information (address, port) and the messages that a service can accept. That way Web Service user does not have to deal with underlying implementation issues. All that user has to know is how to address a service, what parameters to send and what response to expect. This simple object request broker architecture is further simplified by SOAP, which is an XML-based message passing protocol. All communication between Web Services is encoded in SOAP. Users send SOAP requests to Web Services and receive SOAP responses from them. The usual transport protocol chosen for SOAP is HTTP, but this is not mandatory. SOAP messages can be sent over SMTP (Simple Mail

[2] http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211/

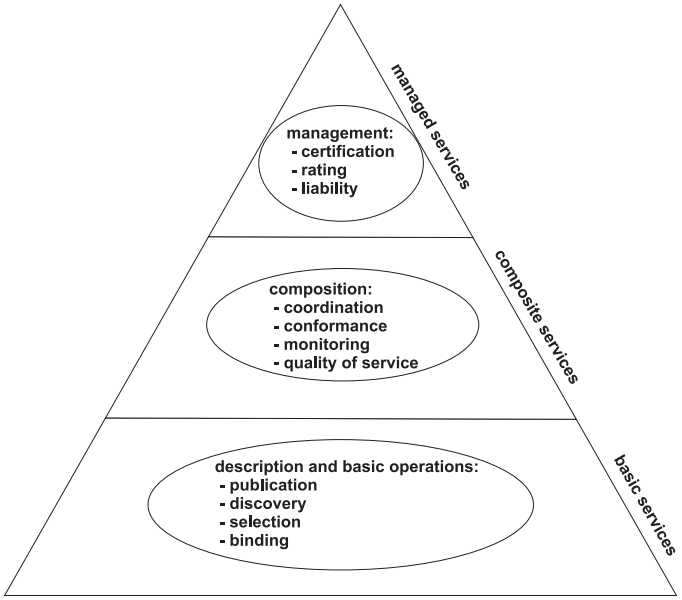

Figure 1.1: Web Services Architecture Stack

Transfer Protocol), JMS (Java Message Service), FTP (File Transfer Protocol), IIOP (Internet Inter-ORB Protocol), etc. Finally, UDDI is a directory protocol based on the notion of organizations offering services. For each organization and service, description is defined in terms of keywords, contacts, and classification. Services can also be searched by names and partial content of their WSDL descriptions, but this requires UDDI customization.

This is in line with requirements postulated in Web Service architecture description, such as logical view of actual system, message orientation, description orientation, granularity, network orientation and platform neutrality. In the next section we will try to incorporate Web Services and SOA in general in the broader context of distributed systems and then question some of the mentioned postulates.

1.3 SOA - Binding Factor of Distributed Systems

The roots of service-orientation can be found in three different areas: programming paradigms, distributed systems technology and business computing [83]. The development of various programming languages provided not only platforms and tools that make SOA possible, but also contributed to bet-

ter understanding of interfacing and interaction problems. Advances in the area of distributed computing, as well as the present ubiquity of the Internet, have also articulated the need for SOA. Although today's distributed technologies, component frameworks and application servers show many striking similarities to SOA, such as uniform interface description through interface description languages (IDLs) or naming services acting as directories (e.g., Java Naming and Directory Interface - JNDI), it is evident that the sheer number of different distributed concepts, standards and products introduced additional problem: *middleware heterogeneity*. Given that technologies such as J2EE or .NET have been initially introduced to solve the problem of application heterogeneity, it is somewhat ironic. Service orientation aims to address many of the problems facing distributed computing today, not only in the area of interoperability but also with respect to the right level of granularity that will initiate wider business to business (B2B) market interactions. Finally, SOA has it roots in the business process modeling and workflow technologies. They both deal with the way business data and logic are processed inside an enterprise. The goal that SOA tries to achieve in this domain is that it leverages business logic and data that is highly heterogeneous and distributed. For the first time SOA offers an ability to map a service directly to a business entity (process) instead of the underlying technical component (e.g., an EJB component).

The true value of SOA however, compared with other software architectures, lies in the fact that it is the first software architecture that in a way transcends software and aims to become general system architecture around which hardware and software systems alike will be built. The prime example is the Open Grid Services Architecture (OGSA) [49], where Web Services have been used by the grid community to develop Grid Services [136]: a mechanism for creating, naming and discovering Grid Service instances. Web Services are also standard way of interconnecting information, resource and service grids by providing abstraction for underlying infrastructure and technologies [137]. On the other hand, the SOA paradigm that emphasizes lightweight collaboration between relatively autonomous and loosely-coupled units (services) is equally applicable to embedded systems inside a car. It has been shown that even the simplest embedded systems (such as sensors) can become parts of SOA, provided a certain supporting infrastructure. [116, 177].

We can further introduce self-descriptive and reusable embedded systems as COTS components into everyday computing environment. These systems can easily interact with communication and computing infrastructure already in place using dynamic and ad-hoc protocols [115]. Computation and communication thus converge, as the worlds of "very small" and "very big" come

together. SOA is a decisive binding factor that can build a bridge between the two worlds, providing hybrid services (e.g., location-based mobile services that are context-sensitive and offered dynamically to the users).

A computing infrastructure where "everything is a service" offers many new system and application possibilities [114]. Among the main challenges, however, is the issue of standardized way of application development in such heterogenous environment. The natural way of doing this is by performing service composition, either by creating services and composing them according to requirements, or (better) reusing existing services in order to archive a given task. Ultimately, the goal is to have an environment of devices and software entities where the demarcation line is somewhat "blurred", and in which composition is performed on-demand, by specifying complex target service description only (automatic service composition).

In such open environment the ability of services to adapt and be extended represents the primary driving force. The main benefits of adopting a composable service infrastructure as the foundation for interdomain collaboration through service composition are facilitated interoperability, standardized discovery, selection and invocation. However, other issues like timeliness, security and dependability are of the paramount importance, too.

1.4 Problem Statement

The first problem we are faced with when investigating Web Service composition is the lack of a valid definition. It seems that many people are working on Web Service composition without bothering to define it. We distinguish between *service orchestration* and *service composition* [111] and define them accordingly:

Def. 1 *Service orchestration is a process of coordinating two or more services, using flow control commands that determine topology, conditions and order of message exchange. It focuses on the behavior (message exchange) of a single participant and defines relevant properties from its perspective only. Terms aggregation and coordination are used as synonyms.*

Def. 2 *Service composition is a process of binding two or more services into a new one using composition operators. Functional and non-functional properties of a new service can be determined and guaranteed, ensuring predictability and correctness of the resulting service's properties. Composition focuses on the global interactions among all participants. Sometimes, term choreography is used instead.*

Automatic (goal-based) service composition is defined:

Def. 3 *Automatic service composition is a composition where only the set of available services and target (goal) service are given. The target is described in terms of functional and non-functional properties it has to fulfill. Automatic composition needs to identify adequate composition using available services and verify that it matches the target description.*

Orchestration is a way to use traditional programming techniques and methods to 'program' services into different paths of execution using flow control commands and has been partially addressed by WS-Coordination specification [29], later adopted into Business Process Execution Language (BPEL) [9] which is further discussed in the Related Work. Service composition, on the other hand, is a process of building a new service out of existing ones, for which we want to guarantee how it will behave with respect to functional properties (what it will deliver) and non-functional properties (how and under which conditions it will deliver). The difference between orchestration and composition is shown in Figure 1.2. Orchestration (Figure 1.2a) describes logical and temporal order of message exchange among different services. Composition (Figure 1.2b) is not concerned with the message level but with the logical patterns in which services can cooperate (such as sequential or parallel compositions). We consider the ability to develop applications using logical constructs (patterns), that are much closer to the way we think, a clear advantage compared with dealing with orchestration at the message level. Furthermore, modeling composition in terms of cooperation models and patterns results in a kind of recursiveness: applying a certain composition pattern to two or more services produces another service which can be further utilized in compositions. Therefore, in this context we are not interested in service orchestration, but in service composition.

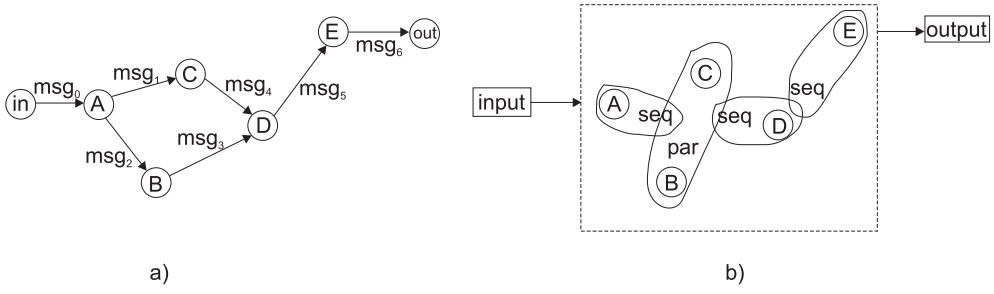

Figure 1.2: Service Orchestration and Composition

A good example of a composite service is an integrated financial management Web Service that comprises more specialized services for payroll, tax preparation and cash management. Choreography of a composite Web Service in this case is defined, for each component service, by a logical sequence of messages that are exchanged among the component services. From the composition perspective we are not interested in the message exchange aspect only, but in the properties of a component and composite services alike and their relationships. The key issue is how to reason about the properties of a composite Web Service based on the declared properties of component services. If a cash management service declares that it can process only those employees that have been previously processed by tax preparation service, and tax preparation service allows for some employees to file tax forms manually, while the payroll service makes no distinction whatsoever who files tax forms, what will be the behavior of the composite system for employees that file their tax forms manually and for those that do that automatically? What will be the result of doing payroll and tax processing in parallel and initiating cash management transaction for an employee whose tax has not been fully processed? Who has the access right to the payroll data and what security credentials are required to initiate cash processing for a given employee? How will the tax preparation service react if there happens to be a crash in the payroll database and the balance becomes inconsistent? Is it possible to compose another service on top of this composition, for example, an automatic bonus system that compares payroll and tax preparation data, and adds a bonus to selected employees? These are some of the questions that choreography approach cannot answer.

The need to capture Quality of Service (QoS) properties in service-oriented architectures is more emphasized than in conventional component middleware [191]. The reason is that services are developed independently of their clients, sometimes by different units within an enterprise and very often even by different organizations altogether. Competition and differentiation of services play the major role in such environment, and knowing QoS properties for a given service offers a possibility to manage different compositions. The reasons why Web Services fail to develop a satisfying composition model so far are:

- Oversimplified description model, that abstracts *too much*. We argue that, while it is beneficial to decouple implementation and specification, it should not be done at the cost of disregarding or ignoring non-functional properties, such as security or timeliness. The bare WSDL description provides hardly enough information to perform composition.

- Poor search capabilities of UDDI limit the extent to which search for partner services (those that participate in composition) can be effectively performed. UDDI was not originally developed as service directory, and this legacy is burdening the architecture today.

- When failing to solve the problems mentioned above, basic Web Service architectures are being augmented with Turing-complete coordination languages that deal more with implementation than with specification, thus creating two additional problems: negating the very basic postulates of SOA, and rendering any kind of correctness verification impossible.

- Since there is no support for non-functional properties, any serious attempt to compose Web Services automatically based only on WSDL properties is very difficult, if not impossible.

There are many additional WS-frameworks and specifications aspiring to become standards, like WS-Addressing, WS-Transactions, or WS-Security, that try to fill in the other missing requirements of Web Services. What is not clear for the present, however, is how they will or even can cooperate with one another. Each solution targets a specific problem not taking into account other requirements. What is currently missing is a unification effort towards WS-Architecture [170]. One possible definition of a software architecture is [48]:

> A software architecture is defined by a configuration of architectural elements - components, connectors, and data - constrained in their relationships in order to achieve a desired set of architectural properties.

A solution to the problems stated above is to develop a set of architectural elements and their relations in SOA such that composability is achieved as a property of the architecture. Our approach is to improve native description capabilities, and to reduce complexity of coordination/composition languages, by challenging one of the basic service-oriented computing's postulates, namely that a service should never disclose any implementation details. It will be argued in the dissertation that disclosing implementation issues up to a certain level not only makes semantic composition and verification possible, but also opens the possibility of truly automated, on-demand service composition. Without disclosing implementation and semantic properties, late (dynamic) service biding, based only on WSDL description, is hardly possible. Based on this premise, a framework for semantic, verifiable and automatic service composition will be developed. Therefore, the contribution

of this work is to identify key SOA elements and develop constraints required for:

- Specification of non-functional properties, by introducing contract-based mechanism for Web Service description (Chapter 3)
- Verification of composition correctness, by providing formal composition language (composition operators) (Chapters 3 and 4)
- Reusable and verifiable architecture by defining basic service design patterns and expressing them using composition operators (Chapter 5)
- Automatic (goal-based) service composition, by treating automatic composition as a search problem and developing and comparing adequate search algorithms (Chapter 6)

In order to demonstrate the approach, a Web Service composition server has been implemented, featuring distributed transaction, exception and state management (Chapter 7).

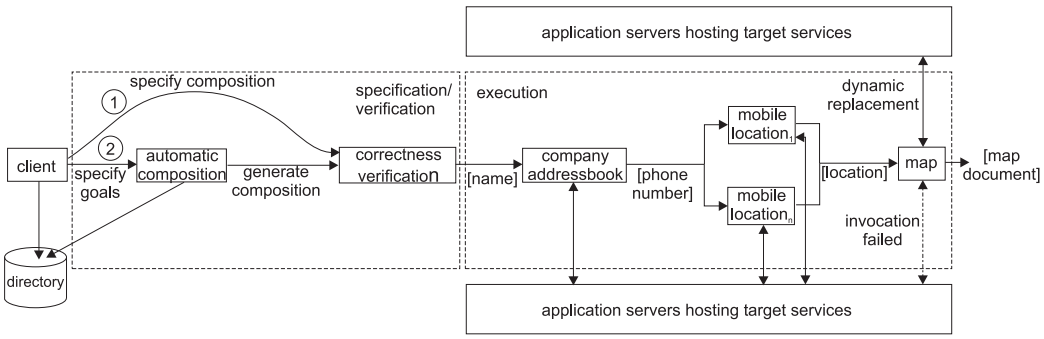

Figure 1.3: Composition Example

Figure 1.3 shows a scenario of service composition, based on the idea of locating a person knowing his/her name [158]. The problem of person location is solved by sending person's name to the address book Web Service, which provides the person's mobile phone number. The phone number is used to query available mobile operators offering Web Services that can return user's current location if the provided number is part of their network. Finally, user location information is sent to the map Web Service, which renders the location in a user-friendly way. The proposed solution enables:

- Manual composition, by browsing service directory, selecting adequate partners (address book service, mobile operators' location services, map

service) and specifying their composition (marked with 1 in Figure 1.3). Subsequent composition verification is automatically performed in order to match whether selected services are functionally and non-functionally compatible.

- Automatic composition, by specifying goal of the composition (person's name is available, and map with the person's current location is required), using functional and non-functional constraints such as price or security (marked with 2 in Figure 1.3). Adequate composition is then automatically generated.

- Dynamic replacement (fail over) in case that some of the services building the composition fail (e.g., a map Web Service fails, another compatible service is dynamically located and incorporated in the composition).

Chapter 2

Related Work

2.1 Basic Composition Requirements

The complexities of distributed systems and increasing trust barriers have influenced the evolution of service-oriented computing (SOC) at several layers: hardware, operating systems and application. Although modern operating systems can also be seen as sets of collaborating services, we focus on the application layer. From the developer's perspective, service composition offers the possibility of reuse. From the user's perspective, composition offers seamless access to a variety of complex services.

Service composition is governed by different requirements than mainstream component based software development. Application developers and users do not have access to documentation or code (either source or binary), but only to a rudimentary functional description offered by WSDL. Services execute in different containers, separated by firewalls and other trust barriers. Therefore, a composition mechanism must satisfy several requirements that we identified in [110]: *connectivity*, *non-functional properties*, *correctness*, *automatic composition* and *scalability*.

Every composition approach must guarantee connectivity. With reliable connectivity, we can determine which services are composed and reason about the input and output messages. However, since Web Services are based on message passing, non-functional properties, such as timeliness, quality of service, security, and dependability must also be addressed. Correctness of composition requires that the properties of the composed service (such as security or dependability) must be verified. Automatic composition is the ability to automatically perform goal-based composition. Finally, since it is likely that complex business transactions will involve many services in a long invocation chain, composition frameworks must scale with the number of the

composed services. The following two examples motivate the need for some of these requirements.

First, suppose we have a trusted and untrusted service, where trust is defined by the service architecture. What happens when we compose these services in sequence? Is this composition trusted, untrusted, or something in between? The composition like this will very likely be untrusted (but not always), so it is crucial that we know whether our service-based application is secure or dependable. And what happens when we compose two trusted services? Do we assume that the composition of trusted services will also be trusted?

Another example that demonstrates the need for addressing non-functional properties is temporal extension of composition (timeliness). Let us observe a simple handshaking example with two partner services, one wanting to invoke a method on another. The client service expects to be notified when it can apply (invoke a method), while the provider service expects to be notified that the client wants to utilize its service. In this way, the composition will not produce useful or expected results, unless we know in advance about handshaking requirements.

Once the native capabilities of Web Services were fully developed, some approaches for service composition started to appear. The first generation composition languages were Web Service Flow Language (WSFL), developed by IBM, and Web Services Choreography Interface (WSCI), developed by BEA Systems. However, these proposals were not compatible with each other, and as the result, the second generation languages were developed. The prime example of these is Business Process Execution Language for Web Services (BPEL4WS), which is a joint effort of IBM, Microsoft and BEA and was realized by combining the first generation languages (WSFL and WSCI) with Microsoft's XLANG specification. In spite of that, the Web Service Architecture Stack still lacks a standard for the *process* layer comprising aggregation, choreography, and composition (www.w3.org/2002/ws/). Therefore, in the remainder of this chapter we cover existing proposals for Web Service composition and compare them with respect to connectivity, non-functional properties, correctness, automatic composition and scalability.

2.2 Business Process Execution Language

BPEL is an XML language that supports process-oriented service composition [37]. It was developed by BEA, IBM, Microsoft, SAP and Siebel (www.ibm.com/developerworks/library/ws-bpel/) and is currently being standardized by the Organization for the Advancement of Structured Information

Standards (OASIS). Recently, Sun Microsystems joined the OASIS technical committee. BPEL composition is a process that interacts with a subset of Web Services in order to achieve given task. The result of BPEL composition is called a process, and participating services are partners. Message exchange or intermediate result transformation is called an activity. Therefore, a process consists of a set of activities. A process has a WSDL interface that enables interaction with partner services. Partner services are external to the process, and all interaction between them is done via WSDL interfaces.

A process is defined using a BPEL source file (.bpel), process interface (.wsdl) and (optionally) a deployment descriptor (.xml). The source file describes activities, the process interface describes ports of composed service, while the deployment descriptor contains the physical location of partner services. The implementation and location of partner services can be changed without modification of the source file.

There are several groups of BPEL elements, and we will list only basic ones:

- starting a process: `<process>`
- defining services participating in composition: `<partnerLink>`
- synchronous and asynchronous calls: `<invoke>`, `<invoke><receive>`
- intermediate variables and results manipulation: `<variable>`, `<assign>`, `<copy>`
- error handling: `<scope>`, `<faultHandlers>`
- sequential and parallel execution: `<sequence>`, `<flow>`
- logic control: `<switch>`

We will model composition of three services using BPEL. Service A is called synchronously and it starts a process. Then, two asynchronous services, B and C, are called in parallel using the output of the first service as their input. The process waits for their completion and then makes a decision based on the results. The stripped BPEL code for this composition would look like this (this is a skeleton with much code omitted for clarity; we assume that all services have only one operation offered at only one port):

```
<process name="test">
  <partnerLinks>
    <partnerLink name="client"/>
    <partnerLink name="serviceA"/>
```

```
  <partnerLink name="serviceB"/>
  <partnerLink name="serviceC"/>
</partnerLinks>
<variables>
  <variable name="procesInput"/>
  <variable name="AInput"/>
  <variable name="AOutput"/>
  <variable name="BCInput"/>
  <variable name="BOutput"/>
  <variable name="COutput"/>
  <variable name="processOutput"/>
  <variable name="AError"/>
</variables>
<sequence>
<receive name="receiveInput" variable="input"/>
  <assign><copy>
    <from variable="processInput"/>
    <to variable="AInput"/>
  </copy></assign>
  <scope>
    <faultHandlers>
      <catch faultName="faultA" faultVariable="AError"/>
    </faultHandlers>
    <sequence>
      <invoke name="invokeA" partnerLink="serviceA"
        inputVariable="AInput" outputVariable="AOutput"/>
    </sequence>
  </scope>
  <assign><copy>
    <from variable="AOutput"/>
    <to variable="BCInput"/>
  </copy></assign>
  <flow>
    <sequence>
      <invoke name="invokeB" partnerLink="serviceB"
        inputVariable="BCInput"/>
      <receive name="receive_invokeB" partnerLink="serviceB"
        variable="BOutput"/>
    </sequence>
    <sequence>
      <invoke name="invokeC" partnerLink="serviceC"
        inputVariable="BCInput"/>
      <receive name="receive_invokeC" partnerLink="serviceC"
        variable="COutput"/>
    </sequence>
  </flow>
  <switch><case>
    <!-- assign value to processOutput -->
  </case></switch>
```

```
    <invoke name="reply" partnerLink="client"
      inputVariable="processOutput"/>
  </sequence>
</process>
```

Recently, a combination of BPEL and Java has appeared, called BPELJ (www-106.ibm.com/developerworks/webservices/library/ws-bpelj/) enabling developers to include Java code inside BPEL code. The so-called 'Java snippets' can be used to perform intermediate transformations such as calculation of values to be inserted into documents, constructing and deconstructing documents using information from other documents and variables, calculating values needed for flow controls, and performing side-effects without the need to create a separate Web Service. Snippets can assume they are running inside a J2EE container. A snippet has access to all variables and partner links that are in scope at the location of the snippet. We can write the `<switch>` construct that we omitted from the previous example using Java snippet:

```
<bpelj:snippet>
  <bpelj:code>
    if (OutputB > OutputC)
      processOutput = outputB;
    else
      processOutput = outputC;
  </bpelj:code>
</bpelj:snippet>
```

BPEL can be used with two other specifications, Web Services Coordination (www-106.ibm.com/developerworks/library/ws-coor/) and Web Services Transaction (www-106.ibm.com/developerworks/library/ws-transpec/). WS-Coordination is used to coordinate the actions of Web Services when a consistent agreement has to be reached on the outcome of service activities. WS-Transactions is used to define transactional behavior of Web Services.

There are several implementations of the BPEL orchestration server, for both J2EE and .NET platforms, such as IBM WebSphere, Collaxa BPEL Server, Microsoft BizTalk, OpenStorm ChoreoServer, Active BPEL.

2.3 Web Service Choreography Definition Language (WS-CDL)

Web Service Choreography Language (WS-CDL) presents an approach which is in a way orthogonal to BPEL. It is currently being promoted as a choreography standard by W3C [75]. It offers a binding between programming

orchestration in BPEL and choreography. In that context, BPEL is treated as a language that specifies behavior of choreography participants, while choreography itself is interested in describing message interchanges between participants that grow from complex interactions while accomplishing a common business goal. BPEL is therefore considered an end-point language like Java (orchestration), and WS-CDL complements BPEL orchestration capabilities with choreography. But what is the difference between orchestration and choreography?

Orchestration (in this approach) focuses on the behavior of a single participant and describes the control flow, variables, events, timeouts and exceptions from its viewpoint. After that, a controller (e.g., BPEL server) enforces execution process by following its definition. On the other hand, choreography is concerned with global, multi-party, long-lived, stateful and coordinated interactions. The key difference is that it does not depend on a centralized controller or server, being a true peer-to-peer solution.

Choreography defines the common observable behavior of participating services. It focuses on global viewpoint which is independent of participants' own viewpoints. Exchanges of information occur only when jointly agreed information driven reactive rules are satisfied. Obviously, process algebra-like languages fit perfectly in the scope to describe such behavior. WS-CDL is a language in which a choreography description is specified.

Figure 2.1 shows principle of choreography language. It specifies general type rules that must be fulfilled before execution can take place. In this example, we used abstract constructs to describe these rules, namely, receiving an input, processing it by service A, and then parallel processing of A's result by services B and C. Naturally, we cannot verify the rules when they are specified in the way we did it. A formal language is required to be able to type processes and verify whether they can execute correctly. For this purpose WS-CDL uses π-calculus, a formalism for mobile processes description which is covered in more detail in Section 2.6.

After choreography is described in this manner, some safety and liveness properties can be verified (e.g., deterministic execution and correct termination). Besides offering choreography primitives expressed in π-calculus, WS-CDL supports definition of supporting information necessary for peer-to-peer Web Services interaction. This information includes typing (information types), identifying and coupling collaborating participants (roles, relationships, participants) and information driven collaborations (channels, channel types, variables, activities, reactions, reuse). Most important among these are roles, channels and variables, as they define how services interact according to choreography rules (exchanging typed variables over channels). WSDL ports and types are mapped into channels and variables to perform

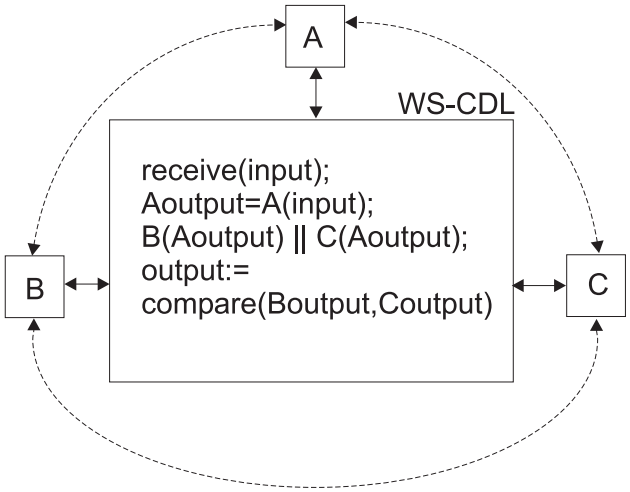

Figure 2.1: WS-CDL Choreography

physical interaction/execution using an XML-based syntax.

2.4 Semantic Web (OWL-S)

Semantic Web is a vision of accessing Web resources not only by keywords, but rather by content [22]. Web Services play an important role in the Semantic Web, since users and software agents should be able to discover, compose and invoke content using complex services. The Darpa Agent Markup Language (DAML) is developed as an extension to XML and Resource Description Framework (RDF) and provides a set of constructs for creating machine readable ontologies and markup information. The part of the DAML program that is related to Semantic Web is the Ontology Web Language - Services: OWL-S (www.daml.org/services/). OWL-S is an ontology of services that enables automatic service discovery, invocation, composition, interoperation and execution monitoring [42]. Note that previous releases of this ontology were called DAML-S, and this name still figures in many papers and reports.

OWL-S models services using an ontology consisting of three parts: service profile, service model and service grounding. A service profile describes what the service requires from users and what it provides to them. A service model specifies how the service works, while service grounding gives information on how to use it.

A subclass of the service model is the process model. A process model describes a service in terms of inputs, outputs, pre-conditions, post-conditions,

and if necessary, its subprocesses. In the process model, we can describe composite processes, their dependencies and interactions. OWL-S distinguishes three types of processes: atomic, simple and composite. An atomic process has no subprocesses. A simple process is not directly invocable and is used as an element of abstraction either for atomic or composite processes. A composite process consists of subprocesses. Constituent processes are specified using flow control constructs: sequence, split, split+join, unordered, choice, if-then-else, iterate, and repeat-until. This is how we could orchestrate the example from the BPEL section using OWL-S (again, only the most important OWL-S commands are shown):

```
<daml:Class rdf:ID="test">
 <daml:subClassOf rdf:resource="Process.CompositeProcess"/>
 <daml:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="Process#composedOf"/>
   <daml:toClass>
    <daml:Class>
     <daml:intersectionOf rdf:parseType="daml:collection">
      <daml:Class rdf:about="process:Sequence">
       <daml:Restriction>
        <daml:onProperty rdf:resource="Process#components"/>
        <daml:toClass>
         <daml:Class>
          <process:listOfInstancesOf
                rdf:parseType="daml:collection">
           <daml:Class rdf:about="#serviceA"/>
           <daml:Class rdf:about="process:Split">
            <daml:Restriction>
             <daml:onProperty rdf:resource="Process#components"/>
             <daml:toClass>
              <daml:Class>
               <process:listOfInstancesOf
                  rdf:parseType="daml:collection">
                <daml:Class rdf:about="#serviceB"/>
                <daml:Class rdf:about="#serviceC"/>
               </process:listOfInstancesOf>
              </daml:Class>
        . . .
</daml:Class>
```

Methods have been proposed for transferring OWL-S descriptions to Prolog [97] and Petri Net-based notation [122] for further analysis related to verification. In the Prolog approach, an OWL-S description is manually translated to Prolog. After that, for a given goal (target) description, it is possible to find adequate plan for composing Web Services. This means

that for a given pool of available Web Services, it is possible to automate the task of finding adequate services that will fulfill the required task using logical inference rules. In the Petri Net approach, an OWL-S description is automatically translated into Petri Nets. This representation is then used to automate tasks such as simulation, validation, verification, composition, and performance analysis.

2.5 Web Component

The Web Component approach [182, 183] argues that services should be treated as components for the purpose of supporting basic software development principles, such as reuse, specialization and extension. The main idea is to encapsulate information about composition logic inside a class definition. The class definition represents a web component. The public interface of a web component can then be published and used for discovery and reuse.

Composition logic is defined inside a class definition of a composed service. Composition logic comprises composition type and message dependency. Composition type can take two forms: order and alternative execution. Order determines whether services will be executed sequentially or in parallel. Alternative execution indicates whether alternative services can be invoked during a process of trying until one succeeds. Message dependency defines input and output message mapping. There are three types of dependency: synthesis, decomposition and mapping. Synthesis composes an output message of a composed service by combining output messages of constituent services. Decomposition binds input messages of the composed service into input messages of constituent services. Message mapping allows custom mapping between inputs and outputs of constituent services.

The basic composition constructs that Web Component supports are: sequential, sequential alternative, parallel with result synchronization, and parallel alternative. They are augmented with condition and while-do. We now show our example in the form of a Web Component class definition:

```
class BC is paraWithSyn{
  public Msg BCInput, BCOutput;
  public oepration(Msg)->Msg;
  private void compose(B.operation, C.operation);
  private void messageDecomposition(BCInput, BInput, CInput);
  private void messageSynthesis(BOutput, COutput, BCOutput);
}

class test us sequ {
  public Msg processInput, processOutput;
```

```
  public operation(Msg)->Msg;
  private void compose(A.operation, BC.operation);
  private void messageDecomposition(processInput, AInput);
  private void messageSynthesis(processOutput, BCOutput);
  private void messageMapping(AOutput, BCInput);
}
```

A Web Component can be specified in two isomorphic forms: a class definition and an XML specification described in Service Composition Specification Language (SCSL). The SCSL specification consists of two parts: the interface of the composite service and the composition logic. We will now demonstrate how composition logic is specified in SCSL for class test only:

```
<construct>
 <composition type="sequ">
  <activity name="A">
   <input message="AInput"/>
   <output message="AOutput"/>
   <performedBy serviceProvider="A"/>
  </activity>
  <activity name="BC">
   <input message="BCInput"/>
   <output message="BCOutput"/>
   <performedBy serviceProvider="BC"/>
  </activity>
  <messageHandling>
   <messageDecomposition>
    <source message="processInput"/>
    <target message="AInput"/>
   </messageDecomposition>
   <messageSynthesis>
    <source message="BCOutput"/>
    <target message="processOutput"/>
   </messageSynthesis>
   <messageMapping>
    <source message="AOutput"/>
    <target message="BCInput"/>
   </messageMapping>
  </messageHandling>
 </composition>
</construct>
```

Web components also support Service Composition Planning Language (SCPL) and Service Composition Execution Graphs (SCEG). They facilitate planning, selection and generalization of the proposed approach by allowing developer to define execution order and dependencies by combining existing web component class definitions using labeling system. That way nesting, substitution, extension and dynamic selection are supported.

Web components offer compatibility and conformance checking. Two services S_1 and S_2 are compatible when S_1 is at least as capable as S_2, and when S_1 can substitute S_2. Service S_1 conforms to service S_2 when S_1 and S_2 can be combined in such a way that the output of S_1 can be taken as input of S_2. With respect to that, in our example service A conforms to B and C, while B and C are compatible.

2.6 Algebraic Process Composition

The idea of algebraic service composition is to introduce much simpler description than those presented so far, and to ensure verification of properties such as safety, liveness or resource management by modeling services as mobile processes. The theory of mobile processes is based on the π-calculus [119].

The complete description of the π-calculus is outside the scope of this chapter, we just present a brief elementary overview. The basic entity of the π-calculus is a process. It can be an empty process, a choice between several I/O operations and their continuations, a parallel composition, a recursive definition, or a recursive invocation. I/O operation can be input (receive) or output(send). For example, $x(y)$ denotes receiving tuple y on channel x, while $\overline{x}[y]$ denotes sending tuple y on channel x. Sequence of actions is specified using dotted notation: $\overline{c}[1,d].d(x,y,z).\overline{c}[x+y+z]$. This describes a process which sends tuple $[1,d]$ on channel c, then receives tuple at channel d whose components are bound to the variables x, y and z, and finally sends the sum of $x+y+z$ to channel c. Parallel process composition is denoted with $A|B$. Several processes may execute in parallel and communicate using compatible channels.

The reason for describing services in such an abstract way is the need to reason about the correctness of their composition [98]. Let us describe our example composition using π-calculus:

$$A(processInput).\overline{B}[AOutput].\overline{C}[AOutput]|$$

$$B(BInput).\overline{out}[BOutput]|C(CInput).\overline{out}[COutput]|$$

$$out(processOutput)$$

Using simple reduction we can see that the only two possible outcomes of this composition are either *processOutput=BOutput* or *processOutput = COutput*, which means that this composition guarantees lock freedom. In a finite number of steps this composition will produce the desired result.

Apart from verifying liveness, we can treat other relevant properties by assigning behavioral types to processes. There are at least two possible ways to perform typing: only the subset of ports are typed, or the entire process is typed. In the first case, we can proscribe the type or shape of data that may be exchanged via two ports. In our example, that would lead to additional limitations on messages. For example, we could require that both *AOutput* and *BInput* follow some pattern (type) in order for reduction $\overline{B}[AOutput]||B(BInput)$ to be possible. In the current example, any kind of message can be exchanged between processes A and B, but if we type the messages (ports), we could limit the exchange. In the second case, the entire process is typed and the notion of type is then a homomorphic image of the process. In many such systems, process and type are synonyms. More details on whole process typing and the way parallel composition is resolved in such systems can be found in [70, 185].

The general question with respect to the algebraic process composition is what to type. Saying too little can result in inability to verify some properties, such as security. On the other hand, saying too much will result in a complexity that will render verification unable or non-practical. The challenge is to find a balance between pure connectivity description (WSDL) and implementation description (e.g., BPEL).

2.7 Petri Nets

Petri Nets are a well-established process modeling approach. A Petri Net is a directed, connected and bipartite graph in which nodes represent places and transitions. Tokens occupy places. When there is at least one token in every place connected to a transition, that transition is enabled. An enabled transition may fire by removing one token from every input place, and depositing one token in each output place.

Services can be modeled as Petri Nets by assigning transitions to methods and places to states [63, 192]. Each service has a Petri Net associated with it that describes service behavior. A Net has one input place and one output place (ports). At any given time, a service can be in one of the following states: not instantiated, ready, running, suspended or completed. After a Net is defined for each service, composition operators are used to perform composition: sequence, alternative (choice), unordered sequence, iteration, parallel with communication, discriminator, selection and refinement. These operators guarantee the closure property. That means that by composing two or more Web Services we produce another service.

Let $\odot$ be sequence operator and $\|_\alpha$ be parallel operator with communica-

tion. Then we can write our example as $A \odot (B \parallel_\alpha C)$. We use parallel with communication to be able to compare and select between outputs of services B and C. Graphically, our service would look like Figure 2.2.

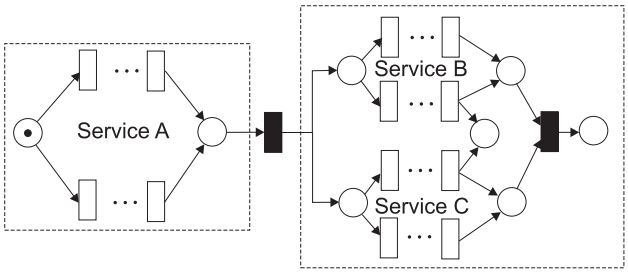

Figure 2.2: Petri Net Composition

This is not the only approach to modeling Web Service composition using Petri Nets. In [165, 167] a workflow language based on Petri Nets is described, while [68, 150] described a methodology for transforming BPEL4WS code to Petri Nets.

2.8 Statechart Composition

In [187, 189, 188] a method is proposed for composing Web Services using statecharts [64]. A composite service is specified using a collection of generic service tasks which are described in terms of ontologies (similar to OWL-S) and then combined according to control- and data-flow dependencies. Statecharts are used to describe these dependencies.

A statechart comprises states and transitions. Transitions are labeled with events, conditions and operations. States can be basic or compound. Basic states (or tasks) are labeled with operation name of a given service class (picked from service ontology). Entering a basic state means invoking operation on a service instance belonging to the given class. Compound states are used for structuring statecharts into regions. Compound states can be either OR-states or AND-states. OR-state contains a single region (sequential execution) while AND-state contains several regions separated by dashed lines (parallel execution). Statechart describing our example is shown in Figure 2.3. Each task in a statechart is annotated with description of non-functional properties. Elementary tasks are evaluated using five generic quality criteria: execution price, execution duration, reputation, successful execution rate and availability.

Based on statechart, an execution path is defined as all possible paths through states (tasks). For every execution path aggregation function computes combined QoS properties. Since each task can be performed by a number of alternative services belonging to the same class, QoS properties are used to perform selection and optimization. Two methods are available: local (task-level) selection of service instances and global planning. The local optimization performs optimal service selection for individual tasks. QoS properties spanning multiple tasks are not considered. On the other hand, global planing uses integer programming to determine composite QoS constraints of a whole service and not individual tasks. Global planning offers possibility of automatic composition based on user's constraint satisfaction. User can specify non-functional properties that need to be satisfied and adequate composition plan can be developed. However, there is no way to verify whether user requirements are consistent, only to measure whether composition plan conforms to the request. Therefore, incorrect request will result in incorrect composition.

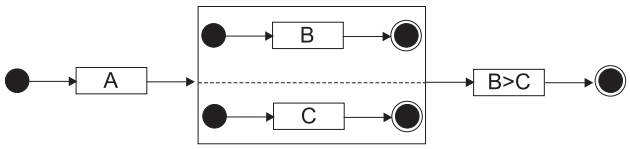

Figure 2.3: Statechart Composition

2.9 Model Checking and State Machines

In this section the remaining approaches for Web Service composition will be described, such as model checking, modeling service composition as Mealy machines, automatic composition of finite state machines (FSMs), usage of abstract state machines (ASM) for adding semantics to BPEL and Temporal Logic of Action (TLA).

Model checking is a method for formal verification of finite-state concurrent systems [128]. System specification is described using temporal logic. The model is then traversed and checked whether the specification holds or not. There are two types of model checking, explicit state and symbolic model checking. Explicit state model checking is related to Buchi automata, a finite state machine that can accept infinite words. In symbolic model checking binary decision diagram is most frequently used to verify finite state systems, and Presburger formulas are used for infinite state systems.

Model checking can be applied to Web Service composition by verifying correctness inside a workflow specification. Among properties that can be checked this way are consistency of data, avoidance of unsafe states (deadlock, liveness), satisfaction of business constraints, etc [51].

Conversation specification has been proposed as another model for Web Service composition [26]. This approach argues that understanding of local behavior of constituent services and global behavior of composed service plays important part in verifying and guaranteeing correctness. Services are modeled as Mealy machines, which is a finite state machine with input and output. Services communicate by sending asynchronous messages and each service has a queue. There is a global "watcher" that keeps track of all messages. The conversation is introduced as a sequence of messages. Then it is shown that by studying and understanding properties of conversations, new approaches for design and analysis of "well-formed" service composition are provided. Using this model it is possible to synthesize constituent service implementations in a top-down approach based on the conversation protocol (desired set of conversations) specified as Buchi automaton and verification of protocol properties [52].

Automatic composition of Web Services is the ultimate goal of most composition efforts. In [21] a framework is described in which behavior of Web Service is described as execution tree and this is then translated into finite state machine. An algorithm is proposed that checks the existence of any composition and returns one if exists. In the process the composition is proved correct. An upper complexity bound is also given.

In [47] and [46] methods are described that use abstract state machines to add formal semantics to BPEL. While the former focuses on using ASM to provide positive (compositional) control flow, the latter provides semantics for handling negative control flow (fault and exception hanlders). Together they provide a comprehensive foundation for BPEL annotation and verification.

Finally, we mention Temporal Logic of Actions (TLA) [84] developed by Leslie Lamport and his work on composing specifications [1]. TLA is a logic for specifying and reasoning about concurrent systems and their properties. Using TLA we can describe Web Services and prove their properties. Also, using theorems for composing specifications from [1] we could reason about properties of composed services. However, we are not aware of any current attempt to use this work to describe composite Web Services.

2.10 Comparative Analysis

We now compare the presented methods with respect to the four service composition requirements: connectivity, non-functional properties, correctness, and scalability. We also discuss the possibilities for automatic composition. Results of the discussion are summarized in Figure 2.4

	connectivity	non-functional	correctness	automatic	scalability
BPEL	√				average
WS-CDL	√		√		very good
OWL-S	√	√			average
Web component	√		√		low
π-calculus	√		√		good
Petri nets	√		√		low
Model checking/FSM	√		√	√	N/A
Statecharts	√	√		√	average

Figure 2.4: Comparison of Composition Methods

2.10.1 Connectivity and Non-functional properties

All approaches offer connectivity of services. While the ways the services themselves are modeled may vary, at the lowest level the connection comes down to the mapping and orchestration of input and output messages between service ports of partner services.

Most approaches neglect specification of non-functional properties, such as security, dependability or quality of service. Only OWL-S allows for definition of some non-functional properties (namely quality of service) but that part is still not completely specified.

2.10.2 Composition Correctness

Verification of correctness depends on the way services and compositions are specified. BPEL and OWL-S do not provide any means to verify correctness. BPEL is a Turing-complete language dealing more with implementation than specification. Therefore, it is difficult to provide any formalism that would be applicable for verification of correctness of BPEL flows. All other approaches support verification in some way. Even OWL-S, when combined with Prolog or Petri Nets, allows reasoning about correctness. However, the extent to which correctness is verified varies.

Web components support simple means to check for compatibility and conformance. π-calculus supports powerful algebraic verification for determining liveness, security, or quality of service. However, it must be noted

that applicability of this verification depends on what is typed when services are modeled as processes. Petri Nets have elaborate algebraic means for verification. We can check whether composition has deadlocks by checking whether corresponding the Petri Net is live and bounded. The model checking approach offers verification methods comparable to π-calculus. One can choose between many available methods for proving that the specification of a composed service conforms to the model. The issue is what needs to be specified in order for model checking to produce useful results. Another problem is that one may simply run out of computing resources (such as CPU time or storage space) and still not know whether the composition conforms to the model because of the vast state space that needs to be examined.

2.10.3 Automatic Composition

Many composition approaches aim to automate composition, which promises faster application development, safer reuse, and facilitates user interaction with complex sets of services. With automatic composition the end user or application developer specifies a goal (a business goal expressed in a description language or mathematical notation), and an "intelligent" composition engine selects adequate services and offers their composition transparently to the user. The main problems are how to identify candidate services, compose them, and verify how closely they match the request. Modeling services as finite state machines promises to be the best way to perform automatic composition so far.

2.10.4 Composition Scalability

All composition approaches support connectivity of Web Services through message passing via ports. However, composing two services is not equivalent to composing ten, or a hundred of them. In a real-world scenario, end users will typically want to interact with more than two services (consider the classic holiday booking scenario), while enterprise applications will invoke chains of possibly several hundred services. Therefore, one of the critical issues is how the proposed approaches scale with the number of services involved. The composition of many services in BPEL is somewhat tedious since XML files start to grow. Since BPEL composition is recursive, one can modularize composition. Unfortunately there is no standard graphical notation for BPEL. Some orchestration servers offer graphical representation, and there are proposals to use UML-like notation for description. Because of the BPEL complexity, graphic notations are not formal, and they do not map 1-1 to BPEL language constructs. Similar issues hold for OWL-S. The Web

Component approach achieves good scalability with class definition, but additional time has to be spent for mapping and synchronizing class definitions and XML. The π-calculus approach offers concise notation with powerful reduction mechanisms, which facilitate specification of complex services. The scalability of the Petri Net approach is reduced by complexity issues, since Petri Nets are not a very scalable modeling technique. Finally, judging scalability of model checkers and finite state machine models depends on the type of the checker and the ways to operate machine states. This discussion is outside the scope of this survey. One should expect that with careful modeling, model checkers can achieve better scalability than Petri Nets and comparable scalability to π-calculus.

2.10.5 Summary

Service-composition approaches that we presented range from those aspiring to become industry standards (BPEL and OWL-S) to more abstract methods. An ideal approach should cover all five key requirements that we identified. The main problem with 'industrial' approaches is correctness verification. Service composition is sometimes called 'programming in the big', yet it seems that industry is unaware that even 'programming in the small' is plagued by numerous problems when formal specification and verification mechanisms are lacking. It cannot be expected that an open paradigm with such varying granularity as Web Services will succeed based on implementation languages alone. On the other hand, formal approaches are often difficult to apply in real-world enterprise environments, and some face scalability problems. From the correctness viewpoint, it is beneficial to analyze Web Service properties using elaborate mathematics; however, to realize these benefits, we must be able to translate from WSDL and SOAP to elegant mathematical solutions.

Chapter 3

Contracts for Web Services

Current standard for Web Service description is Web Service Description Language (WSDL). It allows for description of service input and output parameters, as well as ports and bindings to underlying protocols required for remote operation invocation and data transfer. In this chapter WSDL is augmented with XML-based language that is used for description of service's non-functional properties such as security, dependability or performance. Service description is based on the Design by Contract methodology, hence the language is named Contract Definition Language (CDL). For each Web Service, its pre-conditions, post-conditions and invariants are described using CDL. For the purpose of formal treatment of service contracts, dual (isomorphic) description is adopted. Therefore, service contract is also described using abstract machine notation (AMN). An algorithm for transformation between CDL and AMN has been developed. While CDL description is used to ensure interoperability and seamless network transport, AMN description is used to facilitate formal reasoning about service compositions.

3.1 Design by Contract

Design by Contract (DBC) is a software development methodology, first introduced in [101, 100] and then expanded with basic, behavioral, synchronization and QoS contracts in [23]. It advocates the necessity of standardized and machine readable specification that facilitates component-based development by making reuse safer. DBC is built on three fundamental specification elements: *pre-conditions*, *post-conditions* and *invariants*.

Pre-conditions are component expectations. These are requirements that client must fulfill in order for component to execute correctly. Post-conditions are component guarantees. These are operations (effects) that component

will deliver, assuming that pre-conditions are satisfied. Finally, invariants are static component properties that must be preserved by every component operation. For example, for a component that divides natural numbers, pre-condition is that denominator must not be zero (only if this condition is satisfied, component will execute correctly), post-condition is that result is quotient of input parameters with precision ε (this is what the component will deliver), while invariant is that the result is a rational number (this must never be violated, since division always produces rational numbers).

The proponents of DBC usually refer to the crash of the first Ariane V rocket (June 4, 1996) as to one of the main arguments why it is beneficial to include contracts in software development process [72]. Approximately 40 seconds after launch, the rocket veered off its flight course, then broke up and exploded. The European Space Agency (ESA) established an inquiry board. It eventually determined that the actions of the software inside the on-board Inertial Reference System (IRS) caused the catastrophe. What happened, in effect, was this: the IRS software component controlling one parameter of flight path was reused from Ariane IV project, since its functionality was unchanged. What changed, however, was the range of the input parameters. While Ariane IV could do a roll of say 15 degrees, Ariane V could do 19 degrees. It turned out that the reused component worked quite well for input values up to 15 degrees, but the moment Ariane V rolled more than 15 degrees, unhandled exception was thrown (since the value was physically impossible). This problem could simply be solved by pre-condition mechanism which would catch discrepancy between the reused component and new problem specification. There was a fierce argument between proponents and opponents of DBC whether the rocket could actually be saved by using DBC only [54, 179]. While it is questionable if DBC methodology is enough to solve problems of this kind, it cannot be argued that it is certainly required.

Design by contract can be implemented at two levels: component (service) or architecture level. There are several implementations of DBC methodology at the component level. One of them is Eiffel Programming Language [99, 156]. It is an object-oriented language in which a class definition requires specification of pre-conditions, post-conditions and invariants. More recently, as complexity of software systems started to grow rapidly, other mainstream languages started to adopt some elements of DBC. iContract [39] introduces DBC to Java by adding pre-conditions, post-conditions and class invariants to standard Java classes in form of comments. Prior to class compilation a preprocessor is run which extracts contract from comments and extends class code. Extended class ensures the contract and reports violations. Contract elements are placed inside comments so that a class can be compiled without contract if necessary. This approach is the first of its kind for Java,

but suffers from two drawbacks: it is not mandatory (the class can still be compiled without contract), and it is not possible to specify non-functional requirements (only conditions for input/output parameters). Eiffel is still ahead in this respect, as contracts are mandatory elements of class libraries. It is interesting to note that there are several implementations of DBC for C programming language, used mainly for embedded systems programming [118].

On the other side, architecture-level DBC finds its roots in the architecture description languages (ADLs) [121]. They were introduced to specify high-level compositional view of a software application. ADL focuses on software generation out of deployed components and offers state-transition semantics for analysis and verification of system (application) specification. However, it has been noted [151] that new mission-critical and service-oriented applications developed with modern component based frameworks require additional properties, namely trust and dependability analysis.

Even a short discussion on this topic would be incomplete without Unified Modeling Language (UML)[24]. If nothing else, UML has proven that standardization is necessary in the way complex software systems are designed, specified and developed and evolved . The main downside of UML lies in its complexity and difficult formal treatment which renders it unfeasible for automatic verification. The problem has been recognized and is being addressed by the Object Management Group with Model Driven Architecture (MDA) [60]. However, although MDA and other similar concepts are being developed, the properties we are searching for still remain a wishful thinking.

Our aim is therefore to provide a DBC framework for Web Services addressing both levels of granularity, component and architecture. We propose a solution that is independent of the number of services that are being described and identify properties that are relevant for separate services as well as for complete applications. We accomplish that by treating atomic and composite services equally. In this chapter we will provide definition of contracts for Web Services, and present two isomorphic forms in which they can be represented: XML-based notation and abstract machines. Our main aim, however, is not to use Web Service contracts for reuse only. Instead, our intention is to apply DBC basic ideas in order to develop a contract-based composition framework for Web Services [108]. The way we proceed in this chapter is as follows: we identify what needs to be contained inside a Web Service contract, develop an XML language for description of such service contracts, and then present a mechanism to transfer this description into formal mathematical notation, by modeling service contracts as abstract machines. This notation will be used in the following chapter to perform composition by merging abstract machines (that is, by merging contracts). In the

process of composition, contract of the resulting service is also constructed and verified. It is obvious that for the users of services contrived of a single service (those that do not require composition), contracts will provide the initial goal of DBC, namely, safer reuse.

3.2 Contract Definition Language (CDL)

Our intention is to provide a richer set of descriptive options than those offered by WSDL, since we argue that adequate service description is the key for providing viable composition solution. Specifically, the most serious downside of WSDL is the lack of support for *non-functional* properties. WSDL addresses connectivity problem, but does not allow for specification of other properties that are not directly related to the transformation between input and output parameters (functional properties), but to some internal and external aspects that influence this transformation (non-functional properties). In the domain of Web Services, possible non-functional properties that can be identified are dependability (security, fault-tolerance, real-time), transactions, logging, performance, rendering, etc. There is no way to describe any of these properties using WSDL, yet it is extremely important to know them when trying to build a composite Web Service.

There are two ways to justify QoS requirements for Web Services:

- QoS properties are used in the analysis of overlapping services that offer same functionality. QoS properties provide additional information based on which a comparison and discrimination can be performed.
- QoS properties are used in a process of verification whether two (or more) services are compatible (suitable for a given criteria) for composition.

We propose contracts as additional specification mechanism for Web Service description, side-by-side with WSDL. Contracts include description of both functional and non-functional properties, thus augmenting WSDL. Contracts are specified using Contract Definition Language. We see CDL as an extension of WSDL, and not as a replacement. Therefore, CDL will not include parts of WSDL. Before proceeding to CDL syntax, we will show what WSDL can offer, and what it lacks using an example.

3.2.1 Relationship between WSDL and CDL

Suppose that we want to describe a printing service using WSDL. One way we could do that is as follows:

```
<definitions name="Print" targetNamespace="urn:Foo">
 <types>
  <schema targetNamespace="urn:Foo">
   <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
   <complexType name="Document">
    <sequence>
     <!-- description of Document type -->
    </sequence>
   </complexType>
  </schema>
 </types>
 <message name="PrintIF_print">
  <part name="Document_1" type="tns:Document"/>
 </message>
 <message name="PrintIF_printResponse">
  <part name="status" type="xsd:int"/>
 </message>
 <portType name="PrintIF">
  <operation name="print" parameterOrder="Document_1">
   <input message="tns:PrintIF_print"/>
   <output message="tns:PrintIF_printResponse"/>
  </operation>
 </portType>
 <binding name="PrintIFBinding" type="tns:PrintIF">
  <operation name="print">
   <input>
    <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      use="encoded" namespace="urn:Foo"/>
   </input>
   <output>
    <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      use="encoded" namespace="urn:Foo"/>
   </output>
   <soap:operation soapAction=""/>
  </operation>
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="rpc"/>
 </binding>
  <service name="Print">
   <port name="PrintIFPort" binding="tns:PrintIFBinding">
    <soap:address location="http://localhost:8080/Printer/print"/>
   </port>
  </service>
</definitions>
```

Based on this description, we can only see that service accepts one parameter of type `Document` for method `print` at port `PrintIF`, returns integer variable `status`, using service endpoint `http://localhost:8080/Printer/`. It is quite enough if we just want to connect to this service. However, if we

wanted to compose this service with another, or simply to make a choice between many printing services, it would be much better if we knew something more about the service, e.g., what resolutions, document types and colors does the printer support, how many clients can it serve, will it have enough paper to print the entire document, or how much does it cost to print ten pages? One way that we could describe these and other similar properties is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
  <contract service name="printService" serviceURI="/services/print"
  serviceDescription="Basic printing service" price="0" state="stateless">
  <organization>
   <name>Easy Print</name>
   <description>Printing at affordable price,bw,color</description>
   <classification>
    <type>UNSPSC</type> <value>82.12.15.00</value> <name>Printing</name>
   </classification>
   <primaryContact>
    <name>Mr. Smith</name> <phone>1-800-123456</phone>
    <email>smith@easyprint.com</email> <address>9 Bourbon Street</address>
   </primaryContact>
  </organization>
  <location>
   <country>France</country> <city>Paris</city>
   <street>Rue du Bourbon 12</street>
   <GPS> <latitude>123.456</latitude> <longitude>987.654</longitude>
   <height>100000</height> </GPS>
  </location>
  <method name="Print" methodDescription="Prints a document">
    <parameters>
      <param direction ="in">
        <name>doc</name> <parameterType>Document</parameterType>
      </param>
      <param direction="in">
     <name>resolution</name> <parameterType>int</parameterType>
      </param>
      <param direction="out">
        <name>status</name> <parameterType>int</parameterType>
      </param>
    </parameters>
    <precondition>
      <param>Document.type=ps</param>
      <param>resolution=Printer.res</param>
    </precondition>
    <postcondition>
      <param>Doc.res=resolution</param>
      <performance>
        <type>number-of-concurrent-clients</type>
```

```
        <unit>int</unit> <value>20</value>
      </performance>
      <dependability>
        <transactions model="split">
          <transaction-manager>jrun</transaction-manager>
          <resource-manager>/print/drv/file.drv</resource-manager>
          <compensate-method>printCompensate</compensate-method>
          <timeout unit="ms">1000</timeout> <enlist>required</enlist>
        </transactions>
      </dependability>
    </postcondition>
    <invariant>
      <param>Documen.pages<=Printer.paper</param>
    </invariant>
  </method>
  <event name="outOfPaper"> <!-- event definition --> </event>
</contract>
```

We have described the same printing component in much more detail than we were able using WSDL. We defined organization that provides a service, its location, as well as some non-functional properties, such as document types and resolutions that the printer supports, number of concurrent clients it can serve, transactional model (important if this component is to be enrolled in a complex invocation chain), and events such as printer is out of paper or cannot print the entire document.

Actually, the above example is written in a "loose-form" CDL, with some restrictions omitted for the sake of brevity (e.g, all properties appearing in post-conditions are not explicitly typed in pre-conditions). In the next section we will define strict CDL syntax and semantics in more detail.

3.2.2 CDL Syntax

CDL is an XML-based description language whose purpose it to describe contracts of Web Services. The root structure of CDL is shown in Figure 3.1, while the XSD schema is given in the Appendix A. The root element `contract` comprises `organization`, `types`, `location`, `method`, and `event` child elements. It also has several attributes: `serviceURI`, `serviceName`, `serviceDescription`, `price`, `state`, `version`, and `port`. The names of the attributes are self-describing.

Specifying Organization, Types, Location and Events

Since our aim was to be able to communicate with UDDI directories, CDL retains the notion of organizations and services belonging to organizations (Fig-

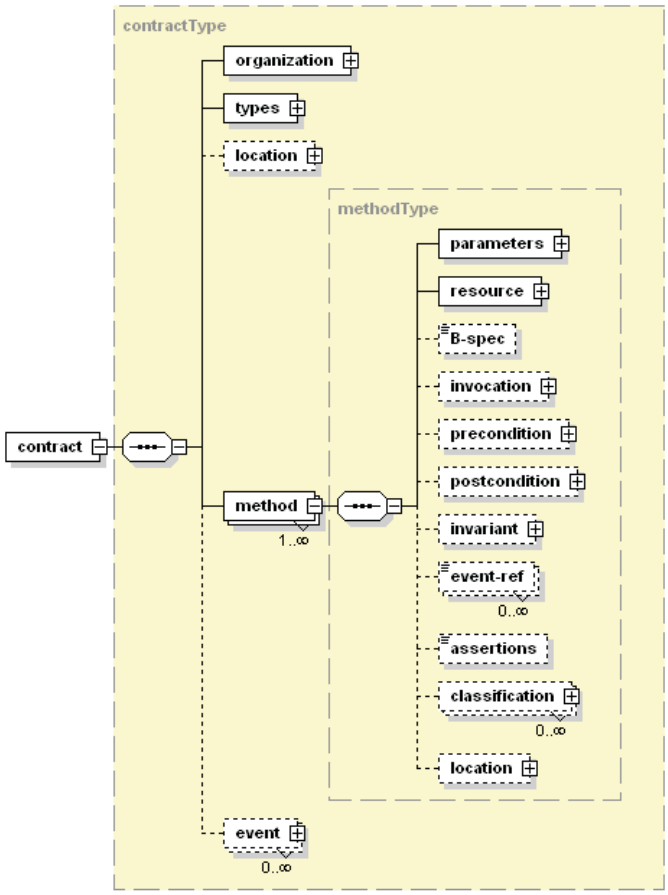

Figure 3.1: The Root Contract Structure

ure 3.2). Organization is characterized by its `name`, `description`, `classification` and `primaryContact` child elements. Name and description are used for keyword-based search. Classification can be performed according to one of the following standards:

- The North American Industry Classification System - NAICS (`http://www.census.gov/epcd/www/naics.html`)
- The Universal Standard Products and Services Classification - UNSPSC (`http://www.eccma.org/unspsc/`)
- The ISO 3166 country codes classification system (`http://www.iso.org/iso/en/prods-services/iso3166ma/`)

However, nothing prevents using other custom classifications, since classification is defined as `type,value` pair. In such cases custom parsing has to be provided.

Primary contact describes a person within organization that can be contacted if human-to-human interaction is for some reason required. Person's name and phone number are mandatory, and email and postal address can be specified optionally.

The `types` element (Figure 3.3) is used when a service accepts or returns complex types, e.g., object of a custom class. In those cases client must know the structure of such complex types in order to be able to construct and deconstruct objects of that class. Service can support more complex types. Therefore, one `targetNamespace` element is defined, and then multiple `complexType` elements, each comprising a sequence of its constituent elements. Elements that build a complex element can also be complex, and are defined in separate `complexType` elements. At the lowest level, all elements must comprise only primitive types from the target namespace.

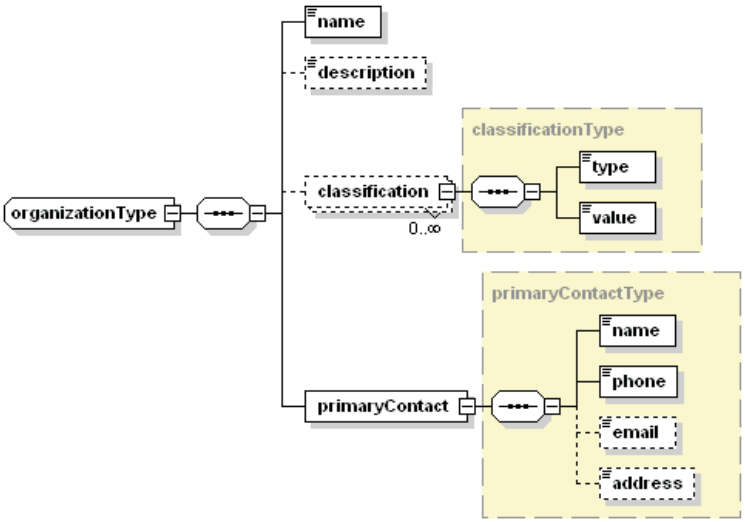

Figure 3.2: Specifying Organization

Location-based services are supported by `location` element. It is an optional element with `country`, `city`, `street` and `GPS` child elements. The `GPS` element allows further specification of `latitude`, `longitude` and `height` of the location offering a service.

Finally, the `event` element specifies events that are supported by all Web methods. It comprises `reference`, `name` and `wrapper` child elements. The

`reference` element is used for unique referencing of events within an organization, `name` is event name within the organization, while `wrapper` is generic wrapper by which event will be known outside organization boundaries. Methods reference particular events that they support via `reference` element.

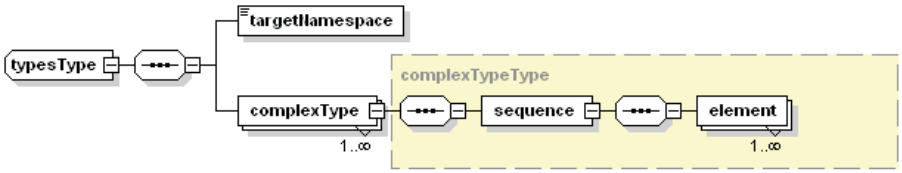

Figure 3.3: Describing Complex Types

Specifying Web Methods

A service can offer one or more methods. They are specified within the `method` element. The root structure of the `method` element is shown in Figure 3.1. It includes `parameters`, `resource`, `invocation`, `precondition`, `postcondition`, `invariant`, `event-ref`, `assertions`, `classification`, and `location` child elements. Apart from them, `method` element has several attributes: `name`, `uri`, `port`, `description`, `price`, and `version`.

The `parameter` element (Figure 3.4) describes method parameters and return values, as well as sets and constants that a service supports. For each parameter it is possible to specify direction (`IN`, `OUT`, or `INOUT`) and whether parameter is required or optional.

The `resource` element is used for connecting with underlying persistent resources for the purpose of state management. More details about handling state are given in Chapter 7. The `invocation` element contains invocation details, such as how to create service instance (if necessary), how to pass a message, and whether a service is involved in synchronous or asynchronous call. With synchronous invocation, client (caller) blocks and waits for the reply, while with asynchronous invocation caller continues with its own execution and is notified when service is completed by polling/callback function. Following are `precondition`, `postcondition` and `invariant` elements. They all share the same structure, shown in Figure 3.5, comprising `log`, `security`, `dependability`, `performance` and `params` child elements.

The `log` element describes properties related to event logging. Locations of `security`, `traffic` and `system` logs can be specified within. The

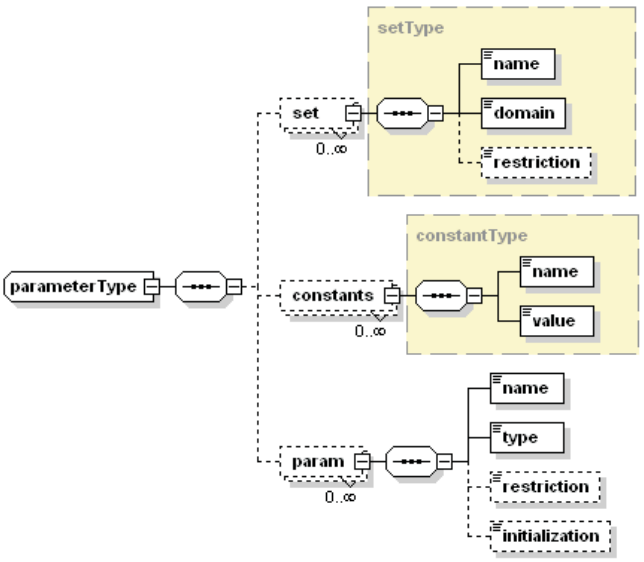

Figure 3.4: Description of Method Parameters

`security` element handles service authentication and authorization. Authentication is described in terms of one or more credential references that must be provided in order to use the service, while authorization comprises on or more security roles. The `dependability` element describes dependability properties of a service and comprises `transactions`, `replication`, `check-point`, `timeout` and `exceptions` child elements.

Transactional behavior is modeled with child elements of the element `transactions`. The `transaction-manager` element is the entity that is responsible for orchestrating transactions, and is usually the container (application server) within which a service is executing. This information is important when complex transactions are built that include services from different containers. The `resource` element is a persistent storage from which a service reads and writes (usually a database). The `resource-manager` helps to connect to an underlying persistent storage (e.g., driver for a relational database). The `compensate-method` is a part of split (open nested) transactional model [58], where one big transaction can be split into a number of smaller ones, that can commit independently. However, if one subtransaction aborts, others that have already committed must compensate. Therefore, for each method that can be involved in a transaction, a service must provide a compensate (undo) method. This model is well-suited for service architectures where it is expensive to lock resources for the duration of the whole transaction, and where transactions can take very long time to finish

[162]. The `timeout` element defines transaction timeout, and `enlist` element describes how a service will enlist in a transaction (whether it supports transactions at all, always requires a new transaction, can join the existing transaction, etc.). Finally, `isolation` element specifies service isolation level. It defines how service expects concurrency control to be performed (read uncommitted, read committed, repeatable read, or serializable).

The `replication` element describes service replication parameters, if service supports replication as means to increase availability. Among properties that are relevant here are `number-of-copies` (describing how many copies of a service exist), `addressing` (how copies are addressed), `broadcast-type`, `ordering`, `delivery`, `response` (the last four describe how to synchronize copies).

The `check-point` element describes how service handles checkpointing. Frequency of making a checkpoint, as well as output file can be specified.

The `timeout` element specifies general service timeout, after which execution will be aborted. This is actually worst case execution time and should not be confused with transaction timeout.

Finally, `exceptions` element describes exceptions that a service method can raise. We distinguish between application and system exceptions, and for each type native (container-specific) and generic (general wrapper) name can be specified.

The two remaining child elements describing pre-conditions, invariants and post-conditions are `performance` and `params`. The former describes service performance, where performance is defined as tuple (type, unit, value). That means that application and/or hardware specific performance can be defined, e.g., (bandwidth, kbs, 20). The latter allows for specification of conditions for parameters, be it pre-conditions for input parameters or guarantees (post-conditions) for output parameters (results).

Going up to the level of the `method` child elements, the remaining elements are `event-ref`, `classification` and `location`. The last two are equivalent to their counterparts at the organization level, and allow for fine-grained description on a method level. If they are omitted, however, classification and location information will be inherited from the parent level. The `event-ref` element refers to one event declared at the level of `contract` child elements, via event's `reference` element. As one method can have more that one event, there can be as many `event-ref` elements as required.

It is clear that all contract elements cannot be filled at design time, especially regarding non-functional properties, such as timeliness or performance. Therefore, contract information is filled gradually, by different roles participating in service lifecycle which will be defined in Chapter 7.

This concludes the syntax of the Contract Definition Language. It should

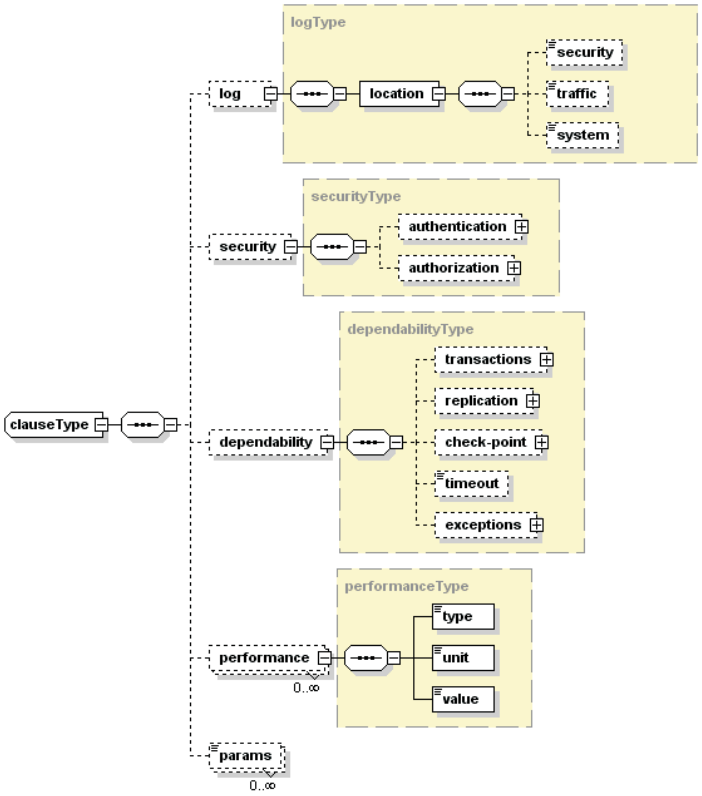

Figure 3.5: Description of Pre-condition, Post-condition or invariant

be clear now what are the critical properties that are specified in a service contract, as well as how to describe them using various CDL constructs.

3.3 Contract Extraction

It can be argued that the proposed specification scheme is too complex, arbitrary and difficult to manage and maintain. In order to deal with this challenge, we show that contracts, as we specify them, can be extracted from the mainstream component frameworks [109], that serve as foundation for service development. Even more, the process of contract extraction can be, to some extent, automated.

The basic premise from which we start the investigation is that specification in terms of contracts is hidden somewhere in the component frameworks that are used today. Therefore, our task in proving that contract option is viable form of specification is to show where the contract elements are hidden

and how to extract them. Some work has already been done in this area, as it has been proven that .NET components harbor hidden contracts which can be extracted by examining library documentation and component source code [10].

We have undertaken a similar effort in order to prove that contracts are hidden inside Java-based components. More specifically, we will look into Java classes and J2EE components (Enterprise Java Beans). Our goal will be to show where to look for hidden specification (one that is not explicitly given in terms of contract elements we defined), and how to extract it (make it available in the form that can be universally used). At the end it will be shown that this process can even be automated to some extent. We believe that this is a sufficient proof that CDL is a realistic way to specify Web Services, since almost all Web Services today are implemented as components running inside either J2EE or .NET containers.

3.3.1 Extraction from Java Classes

The samples from `java.util.ArrayList` class will be used to back our findings, since it is a typical example of a software component (library).

Finding Class Invariants

The following locations have been identified as good candidates to look for class invariants: documentation, constructors, implemented interfaces and base class.

Reading documentation is a logical place to start. However, it immediately confronts us with the first obstacle: it is almost impossible to provide a formal tool for documentation analysis. Therefore, human intervention will be necessary to provide insight on possible invariant candidates. Let us take a look at this paragraph from `ArrayList` documentation:

> Each `ArrayList` instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added in ArrayList, its capacity grows automatically.

From this it can be inferred that `ArrayList` class has an invariant involving property *capacity*. It is equal to the size of the array that is used to store elements in the list. We also find that there is a property *size* and that it is always smaller or equal to the capacity. We cannot infer more about these properties from documentation.

Now we proceed to examine constructors, hoping to find more information on candidate invariants. The class offers three constructors. Inside the first constructor the condition `initialCapacity >= 0` is found, suggesting that a list with negative capacity cannot be constructed. The second constructor creates a default list with capacity 10, which is also non-negative, while the third sets list size to the size of the collection that is passed as an argument. Checking class `java.util.Collection` we find that a collection with negative number of elements cannot be created. This means that invariant property can be established that the size of an `ArrayList` is always non-negative. Since documentation states that capacity of a list is always at least as large as size, implies that we have another invariant: capacity of an `ArrayList` is always non-negative. Establishing an invariant means that all exported methods of a class must preserve this invariant. This conjecture must be proved for every exported method. We do it for only one method, `trimToSize`, that trims the capacity of the `ArrayList` to the current size:

```
public void trimToSize() {
  modCount++;
  int oldCapacity = elementData.length;
  if (size < oldCapacity) {
    Object oldData[]=elementData;
    elementData=new Object[size];
    System.arraycopy(oldData, 0, elementData, 0, size);}
}
```

If the size is equal to the capacity, nothing happens. However, if the size is less than the capacity, the capacity is set to the current size. Since the invariant states that size is always non-negative, it follows that the result of this operation will be non-negative capacity. Therefore, we conclude that this method preserves the invariant. This proof is trivial, but for complex methods can become very tedious.

Next we check implemented interfaces: `List`, `RandomAccess`, `Cloneable`, and `Serializable`. They do not impose additional conflicting requirements on invariants that have been identified. The last thing to verify is the base class, `AbstractList`. In it, however, another invariant property is located, *modCount*, which represents a number of times the list has been structurally modified. Default value for this property is 0, and documentation specifies that it can be only incremented by methods that change list structure. Therefore, the invariant could be: `modCount` is always zero or larger, and must be incremented every time a list structure changes. This is an example how a new invariant can be discovered by checking the base class.

Finding Pre-conditions

Pre-conditions are associated with exported methods, and can be extracted from the following locations: documentation, conditions in exported methods that check input parameters and exception conditions.

In order to discover pre-conditions, exported methods are examined. Documentation can be of help here, but we observed a tendency that pre-conditions are rarely explicitly documented. We consider method `set` of the `java.util.ArrayList` class:

```
public Object set(int index, Object element) {
  RangeCheck(index);
  Object oldValue=elementData[index];
  elementData[index]=element;
  return oldValue;
}
```

The condition for an input parameter is not stated obviously here, but if we look into the body of the `RangeCheck` method, we find it right there:

```
private void RangeCheck(int index) {
  if (index >= size || index <=0)
    throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
}
```

Pre-condition for method `set` is now clear if the condition is reversed: index of an element that is to be set must be less than list size, and greater or equal to zero. Only then the `set` method will be executed correctly. Note that this pre-condition is not even present in the original method, making discovery difficult.

In a language that supports exception handling, pre-conditions are usually coupled with throwing an exception. Therefore, another useful thing would be to look for exceptions thrown by a method, and to reverse conditions that precede exception throwing, or to explore calls inside a `try... catch` blocks that can raise an exception. Using this scheme the favorable conditions for a method can be constructed, under which it will not raise an exception. However, this approach is incomplete. Let us look at the following method signature: `public boolean addAll(Collection c)`. We cannot tell anything about the pre-conditions for parameter `c`. Let us take a look into the method body:

```
modCount++;
int numNew=c.size();
ensureCapacity(size+numNew);
```

```
Iterator e=c.iterator();
for (int=0;i<numNew;i++)
  elementData[size++]=e.next();
return numNew!=0;
```

Still pre-condition cannot be identified using the proposed guidelines (no conditions, no exceptions thrown). However, if this method is called like `someList. addAll(null)`, suddenly a `NullPointerException` is raised. Of course, it is obvious that calling `c.size()` when `c` is `null` will result in an exception, but consider if this was a method with ten parameters and each of them calls a method of its base class. Then the information whether a method can be called with some parameters set to `null` would be invaluable. A solution to this problem is to meticulously examine all possible locations at which an exception can be raised, determine a condition under which it happens, and then to reverse this condition. All this shows that it is tedious and not always possible to spot hidden pre-conditions. Hence the importance of writing pre-conditions a priori.

Finding Post-conditions

Post-conditions describe what a method will guarantee, under the assumption that pre-conditions are ensured. They can be found in documentation and return paths of exported methods.

We observed that the method post-conditions tend to be well documented. Documentation for the method `lastIndexOf` of the `java.util.ArrayList` class says:

> Returns the index of the last occurrence of the specified object in this list; returns -1 if the object is not found.

This is a clear post-condition: if executed correctly, this method will either return index of the last occurrence, or will return -1. This information is crucial for the method caller, and yet it is not the part of a component itself. Sometimes, documentation is not specific about possible outcomes of a method execution, and then we must consider all return paths. We will consider method `indexOf`:

```
public int indexOf(Object elem) {
  if (elem == null) {
    for (int i = 0; i < size; i++)
      if (elementData[i]==null)
        return i;
  } else {
```

```
    for (int i = 0; i < size; i++)
      if (elem.equals(elementData[i]))
        return i; }
    return -1;
}
```

This method has three possible outcomes. Since `ArrayList` can accept `null` elements, this method can return the first occurrence of a `null` element or the first occurrence of a 'regular' (non-`null` element). Otherwise, the method will return -1, signaling that the specified element is not present in the list. Therefore, a way to collect post-condition information is to follow all possible return paths of a method.

Using Javadoc Tags for Contract Extraction

Javadoc comments can be used to extract contract information from Java classes. Javadoc is a tool for generating API documentation in HTML format from comments in source code [103]. It enforces certain coding and commenting practice, that allows for automatic documentation generation. Javadoc tags can be scanned and used for extracting pre-conditions, post-conditions and invariants:

@throws, @exception tags can be used for extracting pre-conditions. They list exceptions that a method can raise. This information can be used instead of scanning method body for possible exception causes. If a constructor is commented with this tag, it can be used to form invariants.

@param tag can be used for extracting method signature. It lists all parameters that a method will accept. This information can be used to form candidates for pre-conditions. If a constructor is commented with this tag, it can be used to form invariants.

@return tag can be used for forming method post-conditions. Javadoc encourages having multiple return values for special cases, which facilitates tracking multiple return paths through a method. Constructors are not commented with this tag, since constructor should always return an object of its own class.

@see tag can be useful in tracking inheritance and dependance behavior of a given class, thus checking if there are conflicting requirements for identified pre-conditions, post-conditions or invariants.

Most tags allow for additional human-readable description, which can be further utilized in discovery of possible contract elements.

3.3.2 Contracts in Enterprise Java Beans

We now expand contract extraction by considering Enterprise Java Beans (EJB). We will quickly revisit types of EJBs. It is important to understand how they differ, because it reflects on locations where we look for contracts for each type.

Session beans model business processes, actions and flow of a system.

Entity beans model data. They cache data from a database inside Java objects and thus represent a developer's window into a database. Using entity beans, developer does not have to deal with various persistence issues, and application servers ensure certain non-functional properties, such as transactional behavior or security.

Message driven beans are similar to session beans in that they represent actions. However, they are called asynchronously, while session beans are called synchronously. The caller does not have to block and wait until message-driven bean responds. It enables communication with services that take long to execute, or are temporarily unavailable.

In the next two subsections, we will try to exploit EJB peculiarities to achieve more effective extraction of functional properties, and to show how to extract non-functional properties. We will not deal with trivial issues such as extraction of component signature from a remote interface.

Finding Pre-conditions, Post-conditions and Invariants

The approach identified in previous sections can be used for extracting pre-conditions, post-conditions and invariants from EJBs, but locations where to look sometimes change. As a case study, proprietary source code has been used, but in this section we use the examples from Java Pet Store, sample J2EE application provided by Sun [106], since it reflects the typical architecture of a business application running on J2EE platform.

Apart from locations already identified, bean pre-conditions, invariants and post-conditions can also be found in: `ejbCreate` and other CRUD (create, read, update, delete) methods, setter methods, primary key classes, finder methods and deployment descriptors.

Knowing whether a component is session, entity or message-driven bean can help in limiting the scope and focusing the search. For session beans, looking into `ejbCreate` method makes sense only for stateful beans, since `ejbCreate` for stateless beans does not accept parameters. For entity beans, `ejbCreate` usually calls setters, so this is the place to look for invariants. Depending on the type of entity bean, bean (bmp) or container managed persistence (cmp), we have several options. Let us take a look at `ejbCreate`

method of `CreditCard` entity bean from Pet Shop example:

```
public Object ejbCreate(CreditCard creditCard) throws CreateException {
  setCardNumber(creditCard.getCardNumber());
  setCardType(creditCard.getCardType());
  setExpiryDate(creditCard.getExpiryDate());
  return null;
}
```

If this is a bean managed persistence component, then `ejbCreate` or setter methods would have to check for parameters, and invariants would be found there. However, if this is container managed persistence component (2.0), setters are abstract, and therefore underlying SQL or EJB-QL (EJB Query Language) statements in deployment descriptor must be examined to try to infer invariants based on cmp fields and associated SQL types, like in this `CreditCard` example:

```
<operation>createTable</operation>
  <sql>
    CREATE TABLE "CreditCardEJBTable" ("__PMPrimaryKey" LONGINT ,
    "__reverse_creditCard___PMPrimaryKey" LONGINT , "cardNumber"
    VARCHAR(255) , "cardType" VARCHAR(255) , "expiryDate" VARCHAR(255),
    CONSTRAINT "pk_CreditCardEJBTabl" PRIMARY KEY ("__PMPrimaryKey") )
  </sql>
```

Based on this it can be inferred that cardNumber, cardType and expiryDate are invariant properties, with length of 255 bytes, and that their length can be zero. Obviously, somewhere in the code an important condition is missing, since this component allows credit card numbers of zero length. This can serve as a good example how deep we sometimes have to go in order to discover contract elements, that should have been clearly specified a priori.

For entity beans, it is good to check constructor of the primary key class, since conditions found there apply to entire entity bean. We also found out that finder methods also harbor hidden contracts, that are otherwise unexpressed. The problem is the same as with `ejbCreate` method: depending on the type of persistence, sometimes it will be required to go to the level of deployment descriptor to extract useful information.

We will also address hierarchy of EJB exceptions and consider which to take into account when scanning for method pre-conditions, since it differs from guidelines we had for general Java classes. Following our recommendations from the previous section, we would have to test for all exceptions that a bean can raise. However, since beans are distributed by definition, it is useful to make a distinction between system-level and application-level exceptions.

Every bean must throw a remote exception, indicating some special error, e.g., network or database failure. These exceptions are of no interest to us when we look for contracts. Sometimes, they are not even propagated all the way back to the client, but can be intercepted by EJB objects that act as a middleware between the client and the bean. Those exceptions are system-level exceptions. Application-level exceptions on the other hand indicate 'regular' problems, such as bad parameters passed to a bean method. Therefore, we must check for all exceptions that are propagated to the client, including all exceptions that the bean defines, and `javax.ejb.CreateException` and `javax.ejb.FindException`.

At the end, message-driven beans will be examined, since many techniques that have been described cannot be applied to them. They have only one, weakly typed business method called `onMessage`. In this method message is decoded and appropriate action is taken. Therefore, specification needs to be provided for one method only. However, this method has no return values, since bean is completely decoupled from the client. That means that post-conditions cannot be inferred based on return types, but since a bean can send a message to the client, this message can be used for finding post-conditions. Message driven beans do not even send exceptions back to the client, in fact, it is prohibited. From this, it is clear that message-driven beans require much different extraction techniques, which is a consequence of asynchronous programming model that they implement. The problem is that message-driven beans do not offer multiple asynchronous methods. They have only one asynchronous method, `onMessage`, which is not a true and general asynchronous model such as asynchronous RMI (Remote Method Invocation). Arguing or speculating how contract information could be extracted from true asynchronous beans, that offer many asynchronous methods, is outside the scope of this work, since such beans do not exist yet.

Extracting Non-functional Properties

In the contract model, the following types of non-functional properties can be described: invocation, security (authentication and authorization), dependability (transactions, checkpointing, replication, exceptions), performance and logging. However, current J2EE specification defines only bean management (lifecycle), persistence, transactions and security. Other functions, such as load-balancing, clustering and logging, are vendor specific and are not considered here. Extracting non-functional properties is relatively simple task, since we look for them in deployment descriptors and application server configuration files. We will now see how to use these elements to form non-functional parts of a contract.

Bean management information is used to form invocation part of a contract. We look into bean deployment descriptor and extract information about remote, home and local interfaces. This information is needed in order to create and destroy beans. Then information whether bean is synchronous or asynchronous is extracted. At the end, for session beans, information on how bean handles states (stateful or stateless) is stored. For entity beans the information about persistence (bean managed or container managed) is used.

The CDL complex type `transactionType` describes transactional behavior and consists of several child elements. Transaction manager is entity that is responsible for orchestrating entire transaction behind the scenes. In this case, it is always the current J2EE container. This element makes sense when we try to reuse or compose components from different application servers. Resource element is a persistent storage from which a component reads and writes data (usually a database). Resource manager helps to connect to an underlying persistent storage (e.g., driver for a relational database). Compensate methods are not supported in J2EE standard. They are part of split (open nested) transactional model [58], where one big transaction can be split into a number of smaller ones, that can commit independently. However, if one subtransaction aborts, others must compensate. Therefore, for each method that can be involved in a transaction, a component must provide a compensate (undo) method. This model is adequate for service architectures, in which it is expensive to lock resources for duration of the whole transaction. Value of `trans-attribute` element is extracted from deployment descriptor. It describes how a component will enlist in a transaction (whether it supports transactions, requires a new transaction, how it joins an existing transaction, etc.). At the end, component isolation level is specified. Unfortunately, there is no easy way to extract isolation information. If a bean is managing transactions itself, we can search for calls like `java.sql.Connection.SetTransactionIsolation()` inside source code. Otherwise, if a container is managing a transaction, isolation information cannot be found in deployment descriptor. We have to look inside application server configuration files or underlying database settings.

Security information encompasses authentication and authorization. An important element in J2EE authentication architecture are the login modules. Each login module implements one authentication mechanism. Therefore, one component can support multiple authentication mechanisms. A list of login modules can be obtained from the configuration module. The name of the configuration module is stored, as well as all login modules. Once the client has been authenticated, it must pass authorization. Information must be exposed that will enable checking whether the client has the right to invoke

certain methods. For that security roles are used. If a bean uses declarative authentication, it is easy to extract security roles from deployment descriptor by reading `security-role-ref`, `role-name` and `role-link` elements. However, if a bean uses programmatic authentication, the source code must be scanned for `getCallerPrincipal()` and `isCallerInRole(roleName)` methods. The first establishes the identity of a client, while the second checks whether it fits in a desired role. By checking all `isCallerInRole` calls, all the roles that a component supports can be identified.

3.3.3 Static and Dynamic Extraction

There are two ways to perform contract extraction: using static or dynamic analysis. Static analysis examines program source code and tries to reason about possible execution outcomes. The model of program execution state is built, e.g., what possible values variables can have. Then it is tracked how they change and specification is inferred. Static analysis is theoretically complete [35], but can be inefficient. On the other hand, dynamic analysis is a runtime analysis of a program. Information is obtained from program executions. Instead of trying to model execution state, actual values that a running program produces are observed. Dynamic analysis is efficient, but it is not general. Therefore, we try to combine the two methods in a hybrid approach, similar to [41]. The process we use is shown in Figure 3.6.

First dynamic analysis is performed, and then refined with static analysis. In the dynamic analysis part, candidate contract elements are identified using heuristics described in previous sections. For that source code, program documentation, information about class inheritance, and language framework in which we operate (access to generic interfaces and base classes) are needed. In order to identify candidates a valid set of test examples must be provided, to cover as many program states as possible during the execution. Since we are trying to obtain formal specification, it is obvious that black box testing cannot be used. Therefore white box testing is employed as we must look inside the code. Statement coverage, branch coverage, extended branch coverage, testing special values and domain partitioning are used, in order to cover as many execution paths so that adequate candidates can be identified. After this step is completed, we try to refine and/or augment identified candidates with static analysis. We inspect code, and try to prove that identified pre-conditions, post-condition and invariants are real. A negation is assumed and proved not possible, or all modeled program states are covered with assumed candidate. In this step additional candidates may be identified.

After this phase a temporary contract is constructed. The next step

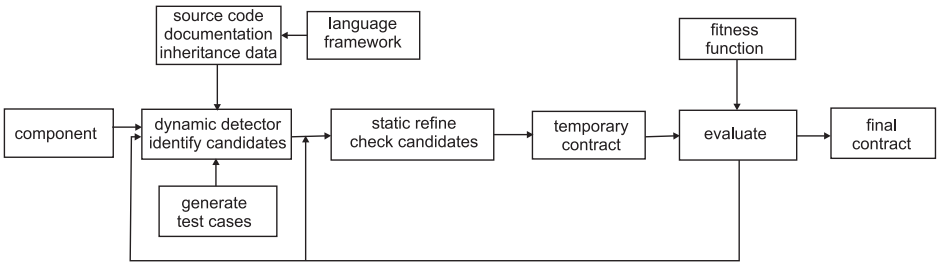

Figure 3.6: Contract Extraction Using Combination of Dynamic and Static Analysis

is evaluation of a temporary contract using fitness function. We test how well a derived specification reflects actual component behavior, trying to predict results for a given test pattern using specification. If a temporary contract is not valid, we can either perform dynamic analysis again, possibly with another set of test cases, or try to determine what is missing using static analysis, by looking deeper into source code or documentation. Once a temporary contract is evaluated as valid, it is promoted into final contract, and used as a component specification.

The corresponding algorithm for contract extraction from typical Java library will now be given. Let $S(CO, MB, IP, OP)$ be the component source code with constructors, method bodies, input and output parameters, *L(I, IF, B)* language framework with inheritance, interface and base class information, $T(t_1...t_n)$ set of test cases, D class documentation, $PS(s_1...s_m)$ model of program states (for static analysis) and $\mathbb{C}$ extracted contract. The same algorithm can be used to extract contracts from complex components (e.g., EJB) by augmenting language framework with necessary information (CRUD methods, finders, deployment descriptors and application server configuration files).

$\mathbb{C} = \oslash$
while ($\mathit{fitness}(\mathbb{C}) = \text{false}$) {

$\mathbf{C} = \mathbf{C} \cup \boldsymbol{invariant(D)}$
$\mathbb{C} = \mathbb{C} \cup invariant(CO, L)$
$\mathbb{C} = \mathbb{C} \cup invariant(IF)$
$\mathbb{C} = \mathbb{C} \cup invariant(B)$
$\mathbf{C} = \mathbf{C} \cup \boldsymbol{pre(D)}$
$\mathbb{C} = \mathbb{C} \cup pre(MB, L)$

$\mathbf{C} = \mathbf{C} \cup \boldsymbol{post(D)}$
$\mathbb{C} = \mathbb{C} \cup post(MB, L)$
foreach (t in T) {
    foreach (c in $\mathbb{C}$) {
        switch(c)
        case (invariant): if !(c, MB, t) $\mathbb{C} = \mathbb{C} \setminus c$
        case (pre): if !(c, IP, t) $\mathbb{C} = \mathbb{C} \setminus c$
        case (post): if !(c, OP, t) $\mathbb{C} = \mathbb{C} \setminus c$
    }
}
foreach (ps in PS) {
    foreach (c in $\mathbb{C}$) {
        switch(c)
        case (invariant): if !(prove(ps,c,MB)) C =
        C \ c
        case (pre): if !$(check(ps, c, IP)$ $\mathbb{C} = \mathbb{C} \setminus c$
        case (post): if !$(check(ps, c, OP)$ $\mathbb{C} = \mathbb{C} \setminus c$
    }
}
}

The main issue in contract extraction is possibility of automation. Some steps of this process cannot be fully automated, and these are represented in bold typeface in the algorithm above. For example, finding invariants in documentation cannot be automated because of the lack of standard documentation format. Another example are J2EE application server configuration files, which are vendor specific and automatization in this area can be achieved on a vendor basis only. On the other hand, inverting conditions that cause exception throwing shows good results when automated, and there are many proposed methods for automatic generation of test cases.

The fact that it is possible to extract contracts in this manner means that contractual behavior is inherent in the way software components are designed and developed, thus justifying our conjecture that contract-based model is adequate description mechanism that is applicable to components building modern Web Services.

3.4 Modeling Contracts as Abstract Machines

If our only goal was to add support for Design by Contract principles to the Web Service architecture stack, extending WSDL with CDL would be the end result. However, our main task is to support not only reuse, but composition of Web Services. As has been discussed earlier, among main requirements that composition mechanism needs to provide are support for description of non-functional properties and verification of correctness. CDL syntax offers a richer set of description primitives compared to WSDL, that can be used for specifying relevant non-functional properties. Verification of composition correctness, however, requires a formal approach.

Since contracts, as they have been presented so far, are just plain XML text files, it would be very difficult, if indeed possible, to judge correctness of their composition. The first problem we would be faced with is actual definition of correctness. What does it mean for a contract to be correct, apart from satisfying XML requirements of being well-defined and well-formed? How can it be judged whether two or more contracts are compatible or not conflicting with each other? How to define relations "compatible" and "conflicting"? Finally, how to perform actual composition when working on text files? In order to be able to answer these questions, a second, isomorphic form for expressing Web Service contracts is introduced: Abstract Machine Notation (AMN). The XML notation is needed in order to transport contracts over a network (interoperability), while AMN serves the purpose of giving contract elements formal mathematical treatment. We first introduce basics of AMN and then show how to map between CDL and AMN.

3.4.1 Introduction to Abstract Machine Notation

Abstract machines are specified and proved using Abstract Machine Notation (AMN). Details on AMN can be found in [2]. Only a brief overview of main AMN principles will be given here.

An element, which can be a class, a component, or a Web Service, is represented as an *abstract machine*. It is characterized by *statics* and *dynamics*. The statics corresponds to the definition of the state, while the dynamics corresponds to the operations. The basic abstract machine elements are specified in the following way:

```
MACHINE M(X,x)
CONSTRAINTS C
CONSTANTS c
SETS S; T={a,b}
```

```
PROPERTIES P
COMPLEX Cx
VARIABLES v
INVARIANT I
ASSERTIONS J
INITIALIZATION U
OPERATIONS
 u1 <- O1(w1) = PRE Q1 THEN V1 END
 ...
 un <- On(wn) = PRE Qn THEN Vn END
END
```

An abstract machine is formally described with several clauses. The `MACHINE` clause defines name of the machine. The name can be parameterized, as is the case in this example. The role of parameters is to leave open a number of finite dimensions of the machine. Parameters can be finite and non-empty independent sets or scalars. Sets are denoted with upper case (`X`) while scalars are denoted in lower case (`x`).

The `CONSTRAINTS` clause allows for definition of constraints that must hold for the parameters of an abstract machine, if they are specified in the `MACHINE` clause. The constraints are expressed in form of conjoined predicates. This clause is used to type scalar parameters, as well as to specify additional constraints on sets, such as set inclusion. However, because of requirement that set parameters are independent, there can be no constraint that makes one set parameter a subset of another one. There is no such restriction on scalar parameters, since a scalar parameter can be a member of one of set parameters.

The `SETS` clause is used to define given sets of an abstract machine. These sets are in fact types. A set is introduced by its name, and then optionally by enumeration of elements. If a set is left unspecified, it is assumed that it is finite and not empty. In the example above, set `S` is left unspecified, and is therefore deferred. Set `T` is fully specified, as it consists of elements `a` and `b`. Similarly to set parameters, given sets must also be independent. The reason is that we use them for independent typing.

The `CONSTANTS` clause defines constants of an abstract machine. They are enumerated in a comma-separated list. A constant can be either a scalar value belonging to some given set, or a subset of a scalar given set, or Cartesian product of given sets. Constants can be given only final values, which cannot be changed afterwards.

The `PROPERTIES` clause binds constants and given sets, by specifying type or value of each constant in the domain of given sets. It is not possible for

formal machine parameter to appear in this clause, as that would lead to possible circularity.

The `COMPLEX` clause defines complex types of abstract machines. Those are the types that can be composed of any combination of scalar types, given set types, subsets of given set types, or Cartesian products of given set types.

The `VARIABLES` clause introduces state variables, which represent components of the machine state. Variables are specified as identifiers in a comma-separated list. Abstract machine performs its operations by changing values of state variables.

The `INVARIANT` clause is used for specification of the invariant property of the state of the machine. It is a property that must be preserved by any operation machine may perform. In other words, `INVARIANT` specifies static laws or rules of a system, and consists of a number of predicates involving state variables. The `INVARIANT` clause must at least provide enough information to allow for the typing of each state variable.

The `ASSERTION` clause is a redundant one as it is deducible from `INVARIANT` and `PROPERTIES`. The reason for introducing this clause is to make abstract machine more readable and to ease formal reasoning that requires use of both clauses. Since invariants type and describe state variables and properties type and describe constants, sometimes it is beneficial to analyse them together, and then it is easier to have certain predicates already deduced, and to use them as additional assumptions in reasoning.

The `INITIALIZATION` clause makes it possible to assign initial values to state variables. These are the values that state variables will have once an instance of abstract machine is created. Later, of course, operations will modify the state, starting from initial values.

The `OPERATIONS` clause defines one or more operations of an abstract machine. Operations express dynamics of the machine. Each operation modifies the state of the machine. This must be done within limits of the invariant, as no operation is allowed to break the invariant. We will be talking about invariant preservation later in much more details. The only way a user of the abstract machine can access the state is using operations. There is no way to access state directly. An operation is defined within `OPERATIONS` clause using operation name (`O1...On`). Operations can have input parameters (`w1...wn`), and can provide return values (`u1...un`). For every operation its body, which is a post-condition it establishes, is defined (`V1...Vn`), along with pre-conditions (`Q1...Qn`). Operation body is specified by means of *substitutions*. Substitution is a change of the state variable value. Next we show what forms of substitution can be used for specifying operation bodies.

3.4.2 Specifying Abstract Machine Operations

Operation body of an abstract machine modifies a machine state. For expressing formally how such modification takes place, we will be using logical predicates relating the values of state variables just before the operation is invoked to the values just after the operation completes. The method used is called before-after predicate. By convention, the values of the variables after the operation has been completed are denoted by priming variable identifiers. Let x be a state variable, and a a scalar value. Then, before-after predicate is:

$$x^{'} = x + a$$

This does not mean that after-value must be related to before-value in a functional way. In general case, this relation can be non-deterministic:

$$x^{'} > x$$

We now generalize before-after predicate by introducing substitution. Let P be a formula, x be a variable and E an expression, then the following denotes the formula obtained by replacing all free occurrences of x in P by E:

$$[x := E]P$$

More details on non-freeness and substitution are given in Appendix B.1 and B.2. Now we show that each before-after predicate can be generalized using substitution. Let us observe the following implication involving before-after predicate and substitution:

$$x \in N \Rightarrow \forall x^{'}(x^{'} = x + a \Rightarrow x^{'} \in N)$$

Applying One Point Rule (Appendix B.3):

$$\forall x(x = E \Rightarrow P) \Leftrightarrow [x := E]P$$

we have:

$$x \in N \Rightarrow [x^{'} := x + a](x^{'} \in N)$$

Performing substitution, we get:

$$x \in N \Rightarrow x + a \in N$$

That is:

$$x \in N \Rightarrow [x := x + a](x \in N)$$

This is a very important result, since it is clear that if we denote invariant $x \in N$ with I and substitution $x := x + a$ with S we can rewrite the previous expression like:

$$I \Rightarrow [S]I$$

This expression says that if the invariant I holds then the substitution S is guaranteed to preserve the same invariant. We now generalize the concept of substitution, by showing different ways to perform it, thus constructing more complex abstract machine operations.

Pre-conditioned Substitution

It has been shown how to denote before-after predicate of the form $x^{'} = x + a$ as substitution, as well as how to express a substitution that preserves a property (invariant) with $S[I]$. We still do not know how to express a very important part of CDL: pre-condition. For that reason pre-conditioned substitution is introduced.

Pre-condition describes conditions under which a service operation will execute correctly, that is, it will reestablish the invariant. Otherwise, the operation will not establish anything. If P is the pre-condition, and S is the substitution guarded by this pre-condition, than pre-conditioned substitution is defined as:

$$[P|S]R \Longleftrightarrow P \wedge [S]R$$

where R is a post-condition that this substitution has to establish. It is obvious that when P does not hold, this substitution will not be guaranteed to establish anything, since $P \wedge [S]R$ will never be true in that case. This substitution can also be presented in the following notation:

```
PRE P THEN S END
```

Multiple Simple Substitution

Often it is necessary to perform simultaneous substitution, where the exact order is not known (cannot or should not be known) in advance. For that purpose multiple simple substitution is introduced:

$$[x, y := E, F]P$$

where variables x and y are being substituted by expressions E and F respectively, in predicate P. This substitution is performed simultaneously and can be defined in terms of simple substitutions:

$$[x, y := E, F]P \iff [x := E][y := F]P$$

If a substitution includes many variables, it can become unreadable. Therefore, it can be expressed in the following (also vertical) notation:

```
x:  = E || y := F ||
      z :  = G
```

Bounded Choice Substitution

Bounded choice substitution is used to express a choice between two or more substitutions. It is nondeterministic in a sense that it is not known which one of them will actually be performed. The results is, however, bounded. This means that whatever the choice is made, the same post-condition must be achieved. If we denote choice between substitutions S and T with $S \Box T$, then the following holds for a post-condition R:

$$[S \Box T]R \iff [S]R \wedge [T]R$$

This substitution can also be presented like:

```
CHOICE S OR T END
```

and is not limited to two substitutions only:

```
CHOICE S OR T OR U OR V OR W END
```

Guarded Substitution

Guarded substitution is similar to pre-conditioned substitution $(P|S)$, as it is also guarded by a predicate. A substitution S guarded by predicate P is denoted:

$$P \Longrightarrow S$$

Substitution S is performed under the assumption P and establishes a post-condition R in the following way:

$$[P \Longrightarrow S]R \iff (P \Longrightarrow [S]R)$$

When P is false, this substitution can establish true or false, and is said to be non-feasible. It should be noted how this substitution differs from pre-conditioned $P \wedge [S]R$. When P does not hold in pre-conditioned substitution, it is said to be aborted, since it cannot establish anything. In guarded substitution, on the contrary, when P is false, the result of the substitution can be anything, as has been shown. A shorthand (also vertical) notation for guarded substitution is:

```
IF P THEN S END
```

Conditional Substitution

Conditional substitution is defined as a combination of bounded choice and guarded substitution in the following way:

$$\texttt{IF P THEN S ELSE T END} \iff (P \Longrightarrow S) \Box (\neg P \Longrightarrow T)$$

or, with post-condition R:

$$[\texttt{IF P THEN S ELSE T END}]R \iff (P \Longrightarrow [S]R) \wedge (\neg P \Longrightarrow [T]R)$$

Unbounded Choice Substitution

The unbounded choice substitution is the generalization of the bounded choice substitution. Let S be a substitution depending on variable z. If we want to specify a substitution that can choose any value of z, we write $@z.S$, where:

$$[@z.S]R \iff \forall z \cdot [S]R$$

Then we can introduce a guard for S and have $@z.(P \Longrightarrow S)$, which can also be denoted as:

```
ANY z WHERE P THEN S END
```

Empty Substitution

The empty substitution does nothing, or more precisely, performs no substitution for the target post-condition. An empty substitution is denoted with `skip`:

$$\texttt{[skip]}R \iff R$$

Obviously, such a substitution is not very useful by itself, but it will be used it to define the while (loop) substitution.

Multiple Generalized Substitution

Multiple simple substitution can be generalized for all substitutions presented so far. We want to enable specification where not only simple substitutions can be performed simultaneously, but where any pair of substitutions can be performed concurrently. The notation remains the same ($\|$), and for any substitutions S, T and U, predicate P and variable z we define:

$$S \parallel \texttt{skip} = S$$
$$S \parallel (P|T) = P|(S \parallel T)$$
$$S \parallel (T \square U) = (S \parallel T) \square (S \parallel U)$$
$$S \parallel (P \Longrightarrow T) = P \Longrightarrow (S \parallel T)^{1}$$
$$S \parallel (@z \cdot T) = @z \cdot (S \parallel T)^{2}$$

Since multiple simple substitution $\|$ is commutative, so is the multiple generalized substitution. Therefore the cases when S is the right side operand will not be shown (e.g., $(P|T) \parallel S = P|(T \parallel S)$).

While Substitution

The while substitution checks for predicate P and if it holds executes generalized substitution S. Then P is checked again, and if it still holds, S is performed again. When P does not hold, the substitution does nothing (skips) and resumes execution right after the loop. This substitution is denoted with:

```
WHILE P DO S END
```

In order to be able to reason about this substitution, another construct is introduced that will describe repetitive substitution. For any generalized substitution T, it is denoted $\hat{T}$ and defined:

$$\hat{T} = (T; \hat{T} \square \ \texttt{skip})$$

Note that this is not a recursive definition. Substitution T is performed completely, and then in the next iteration either the same substitution is performed or nothing is done. Using this formalism, while substitution is defined:

[1] only if $trm(S)$ holds; see Section 4.5.3
[2] only if z is non-free in S

$$\texttt{WHILE P DO S END} \Leftrightarrow (P \Longrightarrow S)^{\frown}\ ;(\neg P \Longrightarrow\ \texttt{skip})$$

This substitution will not be used for constructing operation body, since CDL does not allow for specification of post-conditions in terms of looping. This makes sense, since CDL description should deal with service specification, and not implementation. However, while substitution will be used to construct looping composition pattern in the following chapter.

3.4.3 Why Abstract Machine Notation?

Abstract machine notation (AMN) is not the only formal approach to specification. Furthermore, it is not even the only abstract machine specification, but one flavor among many notations. Since names, underlying formalisms and purposes of these models/notations can be very misleading, sometimes it is difficult to distinguish between them (e.g., abstract machine notation is not equivalent to abstract state machines). Therefore, a short overview of relevant abstract machine and formal specifications is given.

In general, an abstract machine can be defined as a model of a computing system (hardware or software) that is given in terms of its input, output and set of operations (transitions) that lead from input to output. The first and best known abstract machine is a Turing machine. A Turing machine [25] is defined as a tuple $M = (Q, \Gamma, s \in Q, b \in \Gamma, F \subseteq Q, \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\})$. Here Q is a finite set of states, Γ is a finite set of the tape alphabet, s is the initial state, b is the blank symbol, F is the set of accepting states, δ is a partial transition function and L and R are shift-left and shift-right operations. The basic idea of a Turing machine is that it executes a well-defined procedure (algorithm) by changing the contents of an infinite paper tape. The transitions are formulated using action table or transition function, that gives rules how to process any input symbol: for a given state and input symbol, output (write) symbol and next state are given.

Starting from Turing machines, there is an ongoing effort to develop more versatile machines and notations that will be able to represent/simulate algorithms in language independent way. Abstract State Machines (ASM) [28] are not the abstract machines we are using, nor are they specified using AMN. However, they are similar and based on evolving algebras. From the context of Turing machines, ASM try to bridge a gap between computation models and specification methods. Since every computable function can be computed by some Turing machine [164], it is possible to simulate any algorithm by building appropriate Turing machine[3]. This simulation is not

[3]A computable function is the function that can be calculated using a mechanic device.

very appropriate since a single step of an algorithm can require a long sequence of steps of the simulating Turing machine. Evolving algebras (ASM) offer a possibility to simulate an algorithm using a bounded number of steps (transitions). Furthermore, a hierarchy of evolving algebras with different abstraction levels can be used in a description of the same algorithm (operation). They can be used for sequential algorithms (sequential evolving algebras) and for distributed algorithms (although this area is still open-ended). The basic idea is to represent an algorithm as a sequence of states $S(0), S(1), ..., S(n)$ where only a bounded amount of work is done in each transition from state to state. Bounded number of changes (transitions) in state $S(i)$ is performed using rules. In this respect ASM is similar to AMN. States of evolving algebras are static algebras. Static algebra is defined for finite collection of function names as a nonempty set containing interpretations on itself of function names from collection. For each function, its arity is implicitly contained in the function name. For example, for zero-ary functions 0,1 and binary function +, static algebra is a set of integers upon which 0, 1 and + are interpreted in the obvious arithmetic way.

State transition is called programming algebra evolution. It is performed by changing one function at one place using local function update: $f(t_1, ..., t_n) := t_0$. Another type of update is guarded update: *if a then b endif.* A set of guarded updates is executed in parallel and since evolving algebra is deterministic, it is not possible to write $a := true; a := false$ since the order of updates is not known and they contradict each other. In general, rule can be defined as follows:

- any local function update is a rule
- k is a natural number, $b_0, ..., b_k$ are terms and $C_0, ..., C_{k+1}$ are sets of rules then:

  $if b_0$ then C_0

  elseif b_1 then C_1

  ...

  elseif b_k then C_k

  [else C_{k+1}]

It is a partial function of the form $f :\subseteq N \to N$. The algorithm that can be simulated using a Turing machine must be defined using a finite set of instructions dealing with finite number of symbols, and must always execute in a finite number of steps. To avoid potential misunderstandings, it is not true that every *physically* computable function can be simulated using a Turing machine [50], and therefore it is not true that *every* problem can be solved with a Turing machine.

endif

are also rules.

Every rule is equivalent to a set of guarded updates; this is easy to check by induction. Therefore every set of rules is equivalent to a set of guarded updates. A program then is a set of rules. For further comparison between AMN and ASM, [62] features an example of a stack modeled using ASM that can be compared to AMN stack presented in the next section.

ASM theory has recently been applied in creation of Abstract State Machine Language (ASmL) [139]. It is included in Microsoft Visual Studio .NET and can be used for specification and execution of design patterns. That means that design can be verified before the entire application is coded. It is also possible to generate test cases from ASmL model. A model can also be run in parallel with implementation to determine whether implementation conforms to the model.

It can be seen that ASM theory can be mainly used to specify functional behavior of the system, and it is fairly successful in that respect. The key point where ASM differs from AMN is in the way state transition is handled. ASM allows only functional transfer, while AMN is based on the more relaxed set-relational transfer implemented using theory of generalized substitutions. AMN also features pre-conditions, post-conditions and invariants that are used for verification of correctness (proving abstract machine). ASM correctness is based on guarded updates, and while it can be argued that pre-conditions, post-conditions and invariants can be simulated using guarded updates, that would spend many unnecessary steps in abstract state machine execution leading to the same problem that Turing machines are facing: for a single algorithm step many steps of the machine simulating it would be required.

Since AMN is derived directly from the Z Notation, we briefly explain it for the reasons of completeness and also to be able to put AMN in historical context. Z Notation uses schemas to represent specification. Schemas describe statics and dynamics of the system. Static aspects are states that a system can occupy and invariants that must be maintained as system goes through all state transitions. Dynamic aspects are allowed operations and relationships between operations' inputs and outputs. In that sense, schemas are quite close to the notion of abstract machines, both in ASM and AMN. Z Notation uses predicate logic and set theory to describe states and operations, quite similar to AMN. Z schema describing address book that stores birthdays can look like this [157]:

AddressBook

```
persons:  ℙ NAME
birthday:  NAME → DATE
persons = dom(birthday)
```

This defines a schema called `AddressBook`, with two states: `persons` (names of persons stored in address book) and `birthday` (function that maps persons to their birthdays). Then an invariant is given that person must always be an element of the domain of function birthday. The invariant here has the same meaning as in AMN. Allowed operations are defined in the following way:

```
Add
Δ AddressBook
name?:  NAME
date?:  DATE
name?  ∉ persons
birthday' = birthday ∪ {name?  → date?}
```

Operation `Add` modifies state of the schema `AddressBook`, and has two input parameters `name` and `date`, marked with question marks. The first line of the operation body is pre-condition: the name to be added must not be already stored. If this holds, operation takes place by adding new (name,date) pair in the address book. It is clear that Z and AMN share the same foundations. As we saw, apart from minor differences (e.g., in Z pre-conditions can be used only to type parameters, whilst AMN is not restricted in that way), AMN adds to Z the power of expressing operation body using generalized substitution language, which in turn enables elaborate correctness proving, as will be shown in the next chapter.

Finally, Z is evolving too, and has produced an offspring called Object Z Notation [154]. It is an object-oriented extension of Z, allowing for specification of classes and objects as well as inheritance and polymorphism. Instead of generating a sequence of schemas, a system can now be described in terms of interacting objects.

3.4.4 Mapping from CDL to AMN and vice versa

Two notations for describing Web Service contracts have been presented so far: Contract Definition Language and Abstract Machine Notation. During service lifetime there will be times when it will be necessary to switch between them:

- When composing two services, their CDL descriptions will be transferred into abstract machines to allow for formal treatment of their properties.
- When a new service is built (composed), it is constructed by merging abstract machines of the constituent services, thus producing another abstract machine. In order to make this service available to others and to be able to transport its specification over a network, abstract machine has to be transferred into CDL description.

It can be seen that transformation between CDL and AMN has to be bidirectional. However, since this transformation is symmetrical, once we know how to do it one way, the other way is trivial.

The mapping algorithm works as follows:

1. Machine name is constructed from `serviceName` attribute of the element `contract`. All other attributes of the `contract` element, as well as all child elements and attributes of the `organization` and `location` elements are mapped into `CONSTANTS` clause.
2. The `types` element is mapped into `COMPLEX` clause of abstract machine.
3. The `event` element is mapped into `CONSTANTS` clause.
4. For each `method` element, the following is performed:
   (a) State variables set is built from elements specifying properties in `invariant`, `precondition`, `postcondition` elements. All method names and parameters are added to it.
   (b) All sets defined in the `set` element are added to the `SET` clause.
   (c) All constants defined in the `constants` element are added to the `CONSTANTS` clause.
   (d) The `INVARIANT` clause is defined in term of conjoined predicates involving state variables, and is mapped directly from the `invariant` element. The `INVARIANT` clause must contain enough conjuncts to allow for the typing of all state variables.
   (e) The `PRE` clause is mapped directly from `precondition` element. State variables designating input parameters must have constraints (or types) defined in this clause.
   (f) Operation body (postcondition, or `THEN` clause) is constructed by conjoining substitutions from the `postcondition` element. All output parameters must have properties (or types) described in this clause.

(g) All state variables that have `initialization` element defined, are added to the `INITIALIZATION` clause. Additionally, those that are defined as `"INOUT"` are added to the list of machine formal parameters.

(h) The content of `assertion` element (if exists) is added to the `ASSERTION` clause in form of conjoined predicates.

(i) The `resource`, `invocation` and `event-ref` elements are of no interest for composition semantics, and are thus not transferred into AMN. They are used for maintaining internal consistency of composition process, as described in Chapter 7.

The issue that needs to be clarified is treatment of infinite sets (types), e.g., real numbers. Since abstract machines are based on set-theoretical notation, a finite representation for such types is introduced in the following way (taking a set of real numbers as an example):

- The set is limited with `maxReal` and `minReal`
- A finite value ϵ is introduced such that $\forall r_1, r_2 \in R | (|r_1 - r_2| > \epsilon)$

This is machine-dependable, but realistic way to solve this problem, since real numbers are anyway represented with known values of `minReal`, `maxReal` and ϵ for a given target hardware.

The functioning of this algorithm is demonstrated on an example. The complete and formal algorithm description is given in the Appendix C.

Suppose we have a service that offers basic stack operations: push an element to the stack and pop an element from the stack. One way of describing this service in CDL would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<contract serviceURI="localhost:8080/service" serviceName="stack"
port="ServiceIF">
  <types>
    <targetNamespace>urn:test</targetNamespace>
    <complexType name="Performance">
      <sequence>
        <element name="type" type="string"/>
        <element name="unit" type="string"/>
        <element name="value" type="int"/>
      </sequence>
    </complexType>
  </types>
  <method name="push" uri="localhost:8080/service" port="ServiceIF">
    <parameters>
```

```
  <set>
    <name>StackType</name>
    <domain>StackElement</domain>
  </set>
  <set>
    <name>Authentication</name>
    <domain>{Kerberos, Oberon}</domain>
  </set>
  <set>
    <name>Authorization</name>
    <domain>{User, Administrator}</domain>
  </set>
  <constants>
    <name>maxCapacity</name>
    <value>1000</value>
  </constants>
  <param direction="IN">
    <name>element</name>
    <type>StackElement</type>
  </param>
  <param direction="INOUT">
    <name>stack</name>
    <type>StackType</type>
  </param>
  <param direction="INOUT">
    <name>capacity</name>
    <type>int</type>
    <initialization>capacity = maxCapacity</initialization>
  </param>
  <param direction="OUT">
    <name>status</name>
    <type>int</type>
  </param>
</parameters>
<precondition>
  <security>
    <authentication>
      <credential_ref>Kerberos</credential_ref>
    </authentication>
    <authorization>
      <security_role>Administrator</security_role>
    </authorization>
  </security>
  <params>element \in StackElement</params>
  <params>capacity \geq 0</params>
</precondition>
<postcondition>
  <params>stack := stack U element</params>
  <params>capacity := capacity - 1</params>
```

```
    <params>status \in int</params>
  </postcondition>
  <invariant>
    <params>capacity \in (0,maxCapacity)</params>
    <params>stack \in StackType</params>
    <params>status \in N</params>
    <params>authentication \in Authentication</params>
    <params>authorization \in Authorization</params>
    <params>performance \in Performance</params>
  </invariant>
</method>
<method name="pop" uri="localhost:8080/service" port="ServiceIF">
  <parameters>
    <param direction="OUT">
      <name>element</name>
      <type>StackElement</type>
    </param>
    <param direction="INOUT">
      <name>capacity</name>
      <type>int</type>
      <initialization>capacity = maxCapacity</initialization>
    </param>
    <param direction="INOUT">
      <name>stack</name>
      <type>StackType</type>
    </param>
  </parameters>
  <precondition>
    <params>capacity \leq maxCapacity</params>
  </precondition>
  <postcondition>
    <performance>
      <type>WCET</type>
      <unit>ms</unit>
      <value>10</value>
    </performance>
    <params>stack := stack \ element</params>
    <params>capacity := capacity +1</params>
    <params>element \in StackElement</params>
  </postcondition>
  <invariant>
    <params>capacity \in (0,maxCapacity)</params>
    <params>stack \in StackType</params>
    <params>status \in N</params>
    <params>authentication \in Authentication</params>
    <params>authorization \in Authorization</params>
    <params>performance \in Performance</params>
  </invariant>
</method>
```

```
</contract>
```

A stack has been described with capacity `maxCapacity`, that can store elements of the type `StackElement`. It offers two operations, `push` and `pop`. The `push operation` will be executed if there is a space left on the stack and if user can supply `Administrator` credentials which are required for modifying the stack. As a post-condition, the operation will put the passed value on top of the stack and decrement stack capacity. The `pop` operation will be executed if there is at least one element on the stack, and will remove the element from top of the stack and return it to user with guaranteed execution time of no more than 10 milliseconds. Applying the mapping algorithm, the following abstract machine that describes the same service is obtained:

```
MACHINE stack (capacity)
CONSTANTS maxCapacity, serviceURI="localhost:8080/service"
serviceName="stack" port="ServiceIF"
PROPERTIES maxCapacity = 1000
CONSTRAINTS capacity = maxCapacity
SETS StackType(StackElement), Authentication = {Kerberos, Oberon}
Authorization = {User, Administrator}
COMPLEX Performance(string,string,int)
VARIABLES element:INOUT, stack:INOUT, capacity:INOUT, status:OUT,
authentication, authorization, performance, push, pop
INVARIANT capacity ∈ (0, maxCapacity) ∧ stack ∈ StackType ∧
status ∈ N ∧ authentication ∈ Authentication ∧
authorization ∈ Authorization ∧ performance ∈ Performance ∧
push ∈ StackElement → N ∧ pop ∈ {} → StackElement
OPERATIONS
status <- pushToStack(element)
PRE element = int ∧ capacity > 0 ∧
authentication = Kerberos ∧ authorization = Administrator THEN
stack := stack ∪ element ∧ capacity := capacity -1 ∧
status := push(element) END
element <- popFromStack()
PRE capacity < maxCapacity THEN
element := pop() ∧ stack := stack \ element ∧
capacity := capacity + 1 ∧ performance := (wcet,ms,10) END
END
```

All the basic elements required for service composition have now been developed. In the next chapter it will be shown how to compose abstract machines representing Web Services and verify the correctness of their composition.

Chapter 4

Composable Service Architecture

In this chapter elements of the composable service architecture are defined. The basic composition operators are introduced. Composition operators are applied to abstract machines describing services that are to be composed. Each operator is associated with a set of rules that govern how new, composite abstract machine is created based on the abstract machines describing services that are being composed. Composition operators therefore determine composition control flow. Additional data flow options allow to connect outputs and inputs of composition partners in a given way (e.g., aggregating two outputs to one input). After a composite abstract machine has been created, its correctness is formally verified. The process of correctness verification comprises type checking, invariant preservation proofs and correct termination proofs. Using correctness proofs, composable service architecture is defined as an architecture where composition correctness is a safety property, effectively prohibiting composition of incorrect services.

4.1 Composition Patterns

We proceed by identifying five basic patterns to compose services: sequence, parallel, selection, choice and loop. We show how to construct composite abstract machine clauses for each case and then discuss what are the general and specific properties of those patterns. For each composition pattern, adequate composition operator is introduced. Composition patterns therefore should not be confused with *design* patterns described in Chapter 5. In all subsequent definitions, P denotes pre-conditions, S post-conditions (effect substitutions) and I invariants.

4.1.1 Sequential Composition

The sequence operator executes two (or more) services in an ordered sequence. Sequential composition of services A and B is denoted with:

$$C = A \rhd B$$

Outputs of the left operand (A) become inputs of the right operand (B). Therefore, operation $\rhd$ is not commutative. Figure 4.1 shows this composition.

The clauses of the resulting abstract machine are calculated in the following way:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are conjuncted
- `OPERATIONS` clause is constructed by performing substitution of the left operand, then substituting input state variables of the right operand with the output state variables of the left operand, then performing substitution of the right operand, while conjuncting preconditions:

$$\begin{array}{c} \texttt{OPERATION}\ output_B \leftarrow \texttt{C}(input_A) \\ \texttt{PRE}\ \ P_A \wedge P_B \\ \texttt{THEN}\ \ S_A\ \texttt{;}\ input_B := output_A\ \texttt{;}\ S_B\ \ \texttt{END} \end{array}$$

  Here $output_B$ is a set of output state variables of the right operand, $input_A$ is a set of input state variables of the left operand, `C` is the name of a new (composite) operation, $input_B$ is a set of input state variables of the right operand, and $output_A$ is the set of output state variables of the left operand. Naturally, mapping from $input_A$ to $input_B$ has to be provided, unless it is clear that only one mapping is possible.

- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

4.1.2 Parallel Composition

Parallel composition executes two (or more) services concurrently. Two subtypes of this pattern are allowed: parallel composition with and without communication. In the former case, concurrent services can communicate with each other, for the purpose of synchronization of some state variables.

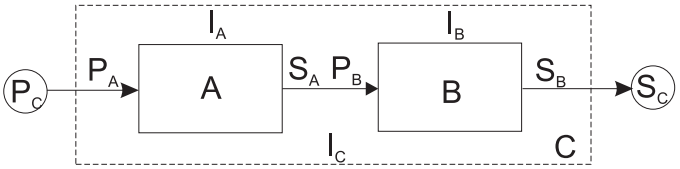

Figure 4.1: Sequence Pattern

It can be used when a certain decision has to be reached after parallel processing has been performed, e.g., choosing result of one service and discarding the other. Only operators of the relational algebra are allowed for the state variables upon which the synchronization is performed. Result aggregation of any kind is not allowed, since it would needlessly complicate composition pattern. If data aggregation needs to be performed, additional service should be created and then sequentially composed to the parallel composition. In the latter case (no communication), there is no communication / synchronization between concurrent services. Figure 4.2a shows parallel composition without communication and Figure 4.2b shows parallel composition with communication.

Parallel composition with communication is denoted with $\|_{P(c)}$, where c are state variables that are being used for synchronization and P is the predicate evaluated upon them, while parallel composition without communication is denoted with $\|$ only:

$$C = A \|_{P(c)} B$$

$$C = A \| B$$

The clauses of the composed abstract machine are constructed in the following way:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are conjuncted
- `OPERATIONS` clause is constructed differently for composition with and without communication:
  - For parallel composition without communication, pre-conditions are conjuncted and substitutions are performed simultaneously (using multiple general substitution):

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ \ P_A \wedge P_B$$
$$\texttt{THEN}\ \ S_A \parallel S_B\ \ \texttt{END}$$

Here $output_C = output_A \cup output_B$ and $input_C = input_A \cup input_B$.

– For parallel composition with communication, pre-conditions are conjuncted and substitutions are performed simultaneously. Afterwards, predicate P is evaluated on a subset of state variables c, resulting in choice of output of only one service:

$$\texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C)$$
$$\texttt{PRE}\ \ P_A \wedge P_B$$
$$\texttt{THEN}\ \ S_A \parallel S_B$$
$$\texttt{IF}\ \ P_c\ \ \texttt{THEN}\ \ output_C = output_A\ \ \texttt{ELSE}\ \ output_C = output_B\ \ \texttt{END}$$

- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

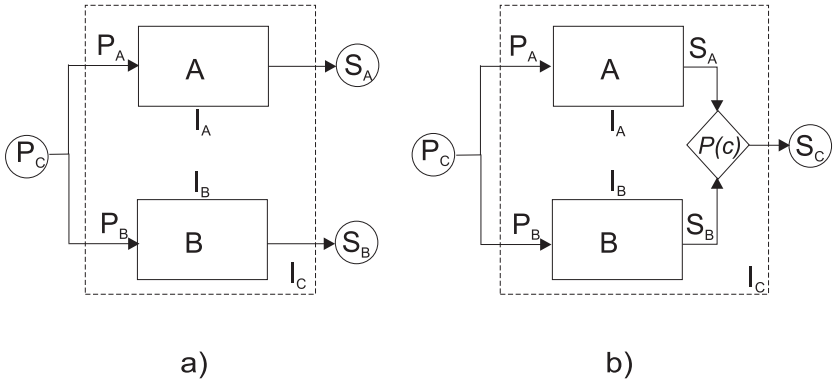

Figure 4.2: Parallel Patterns

4.1.3 Selection Composition

The selection pattern is similar to parallel composition: from the pool of available services, it selects one, without executing the others. Based on the external predicate $C(x)$, this operator selects one candidate service and executes it. Contrary to the parallel composition, selection is performed before execution, thus eliminating concurrent (parallel) execution. This operator can be simulated using parallel communication with communication

by supplying $C(x)$ as a synchronization predicate. This is, however, not recommended since it results in an unnecessary loss of resources such as time (for execution and synchronization), network traffic and money (if service invocation or other resource usage is being charged). This composition pattern is denoted with $\odot_{C(x)}$:

$$C = A \odot_{C(x)} B$$

The composition is shown in Figure 4.3. The machine resulting from application of selection is constructed as follows:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are disjuncted
- `OPERATIONS` clause is constructed by disjuncting pre-conditions, evaluating predicate $C(x)$ and selecting one substitution based on the evaluation result:

$$\begin{array}{c} \texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C) \\ \texttt{PRE}\ \ P_A \vee P_B \\ \texttt{THEN}\ \ \text{IF}\ C(x) \\ S_A\texttt{;}\ \ output_C = output_A\ \ \texttt{ELSE} \\ S_B\texttt{;}\ \ output_C = output_B\ \ \texttt{END} \end{array}$$

- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

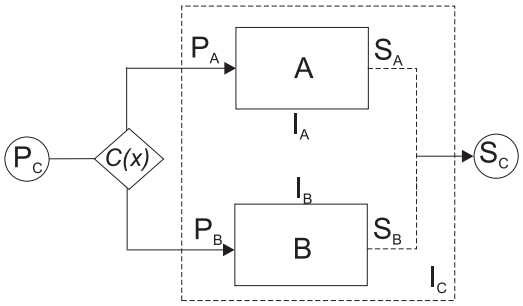

Figure 4.3: Selection Pattern

4.1.4 Choice Composition

The choice pattern represents a composition that behaves as either of its constituent services. It is similar to parallel composition pattern with communication, but it is non-deterministic. It is furthermore restricted to compatible services in sense of input parameters and effects, because it is used when it is known in advance that some of the available services can perform the requested operation, without the need to know which one will do so in a particular instance. The most general example is sending the same request to many services and accepting the results from the first one that completes its execution. In general, this operator is used to express that *any* of the listed operands can fulfill the client's request. This composition pattern is denoted with $\Box$:

$$C = A \Box B$$

The composition is shown in Figure 4.4. The machine resulting from applying choice pattern is constructed as follows:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated
- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are disjuncted
- `OPERATIONS` clause is constructed by disjoining preconditions and connecting substitutions by bounded choice substitution operator:

$$\begin{array}{c} \texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C) \\ \texttt{PRE}\ \ P_A \vee P_B \\ \texttt{THEN}\ S_A \Box S_B\ \texttt{END} \end{array}$$

  Here $output_C = output_A \vee output_B$, which is implied in $S_A \Box S_B$.

- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

4.1.5 Looping

Looping pattern supports execution of the same service repeatedly, until a certain condition is fulfilled. Based on the condition controlling the loop, unary and binary loop are defined:

$$C = \circlearrowleft_{P(e)} A(e)$$

$$C = W(e) \circlearrowleft_{P(e)} A$$

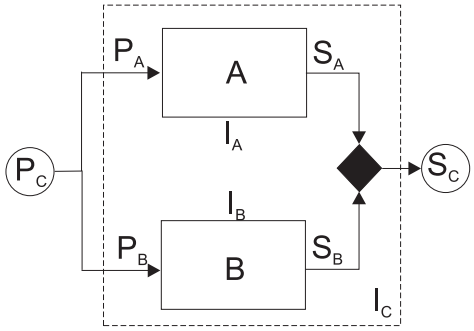

Figure 4.4: Choice Pattern

In both cases, looping is controlled by predicate P evaluated on the variable e. Service is executed until $P(e)$ becomes false. In the unary pattern, e is a state variable of service A, and is changed in every iteration by the execution of A. Therefore, service A controls the loop exit condition. Since this is the loop with the condition on top (exit condition is evaluated prior to execution), variable e must be in the `INITIALIZATION` clause to enable the first loop iteration. In the binary pattern, there is another service W that controls $P(e)$. In this case service A is not allowed to influence the loop exit condition. Here, service W is executed prior to A and will set value of e, which therefore does not have to be initialized. Unary (a) and binary (b) pattern are shown in Figure 4.5.

The composite machine is constructed as follows for the unary pattern:

- The clauses `SETS`, `CONSTANTS`, `VARIABLES`, `PROPERTIES`, `INVARIANT`, `ASSERTION`, and `INITIALIZATION` are kept unchanged. Variable controlling loop exit (e) must appear in the `INITIALIZATION` clause.
- Operation body is constructed by enclosing original substitution in a `WHILE DO` block, controlled by $P(e)$:

$$\begin{array}{c} \texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C) \\ \texttt{PRE}\ P_A \\ \texttt{THEN WHILE}\ P(e)\ \ S_A(e)\ \ \texttt{END} \\ \texttt{END} \end{array}$$

  Here $ouptut_C = output_A$ and $input_C = input_A$.

For the binary pattern, another service W controls exit variable:

- `SETS`, `CONSTANTS`, and `VARIABLES` clauses are concatenated

- `PROPERTIES`, `INVARIANT`, and `ASSERTION` clauses are conjuncted
- Operation body is constructed by conjoining preconditions, and enclosing both substitutions inside a `WHILE DO`:

$$\begin{array}{c} \texttt{OPERATION}\ output_C \leftarrow \texttt{C}(input_C) \\ \texttt{PRE}\ \ P_A \wedge P_W \\ \texttt{THEN}\ S_W(e) \\ \texttt{WHILE}\ P_e \\ S_A; S_W(e)\ \texttt{END} \\ \texttt{END} \end{array}$$

- `INITIALIZATION` clauses are concatenated, and multiple composed if needed

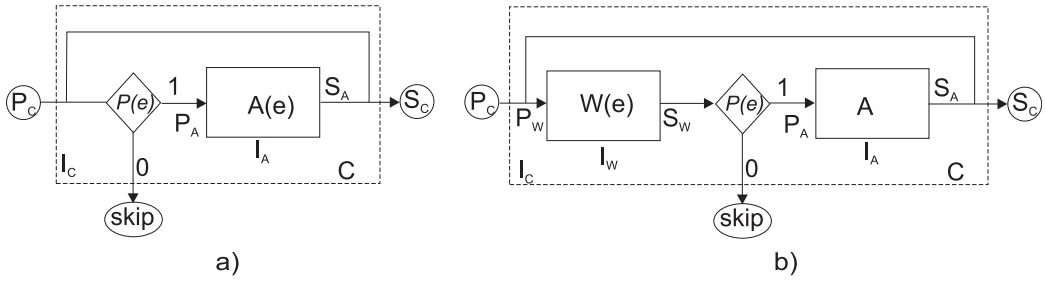

Figure 4.5: Loop Pattern

4.2 Data Flow

In the previous sections, control flow was developed that describes order of the elements (services) inside a composition. Data flow defines how data elements (parameters and results) are exchanged between the elements of composition. A mapping relation is introduced for binding set of available data elements and set of available input parameters. Available data can be either input parameters of the entire composition (external data) or intermediate composition results (output parameters of the previously executed services). Mapping relation effectively establishes which data elements are expected at (passed to) input ports and can be performed in several ways:

- *Direct mapping* is given explicitly for all (available data, input port) pairs. Furthermore, this mapping can be:

  - 1-1, where exactly one available variable is passed to one input variable.
  - 1-many, where one available variable can be passed to more than one input variable (e.g., when a result of one service has to be provided to several services for parallel execution).
  - subset, where only some of the available variables are used, while others are discarded (e.g., when the next composition element requires only partial result of the previous partner).

- *Default (indirect) mapping* where no explicit mapping is given and available variables and input ports are paired on the first-fit basis. This mapping is useful when the mapping is obvious, that is, only one correct mapping exist.

Note that the state of the composition at a given point is determined by sampling all available data (external inputs and service outputs) together, therefore there is no need to provide aggregation of available data, since it is irrelevant and independent of the data origin (e.g., when more services complete in a given step, all their output variables can be explicitly passed to another service using direct mapping).

Mapping relation must satisfy several correctness requirements, namely that all required input variables must have their values properly assigned, and that types of input variables and available variables assigned to them must compatible(see Section 4.5.1).

All data is stored at the composition level using blackboard approach. Composition input parameters are stored in the central storage, as well as intermediate service outputs. That means that if output of one service is used as input for two others, the value must be written once (when the first service completes) and read twice (one per invocation of each subsequent service). This mechanism is similar to the BPEL process variables: intermediate variables will be created for all necessary input and output data. This approach differs from using data connectors [89], where explicit flow connectors connect the output of one activity with the input of another one. Blackboard approach offers greater flexibility and easier implementation, although data flow is admittedly easier to follow when using flow connectors.

4.3 Additional Knowledge and Minimization

Further refinement of the composite abstract machine can be performed in the process of minimization. It introduces additional semantics to the composition process by enabling transformation of states according to behavior

rules. Some states may be deleted, compacted and new states can be generated. Suppose two abstract machines M_1 and M_2 are being composed having state variables s_1 and s_2. We are interested what additional semantics can be applied to s_1 and s_2 *after* the composition process has been performed using given composition pattern. State variables may (but are not obliged to) belong to one of the following behavior classes [176]:

- If a state variable remains unchanged after the composition, it is *invariant* (both s_1 and s_2 still identify properties of S_1 and S_2 respectively).
- If a state variable still exists after the composition, but its value has changed, it is *bounded* (both s_1 and s_2 still identify properties of S_1 and S_2 but their values have changed).
- If a state variable does no longer exist after the composition, it is *vanishing* (s_1 or s_2 or both do not exist in a composite abstract machine after composition).
- If a state variable can generate new states, it is *emerging* (s_1 and s_2 can potentially generate new state variable s_3 and remain in the composite machine themselves).
- Finally, if a state variable vanishes, but together with another state variable can generate emergent state, it is *transferred* (s_1 and s_2 are removed from the composite machine, but on account of that state variable s_3 is created).

Behavior rules are assigned to composition patterns and they determine how to process a given behavior class (e.g., how to generate a new state based on two removed ones). If a behavior class is not specified for a given state variable, no additional transformation will be performed upon resulting abstract machine. Minimization is performed if state variables have a behavior class assigned and composition pattern offers more platform and/or application specific knowledge (rules) on that behavior. Minimization is a way to tailor the properties in a general composition process to specific and dynamically changing environment in which services may execute. Sometimes process of minimization is necessary in order to perform meaningful and useful composition. Let us observe an example where two services S_1 and S_2 are composed in sequence and in parallel. Both declare worst case execution time:

```
MACHINE S1
VARIABLES wcet1
```

```
INITIALIZATION wcet1=1000
OPERATION operationS1
...
END

MACHINE S2
VARIABLES wcet2
INITIALIZATION wcet2=500
OPERATION operationS2
...
END
```

When composed in sequence or parallel, the resulting machines will be:

```
MACHINE Seq
VARIABLES wcet1,wcet2
INITIALIZATION wcet1=1000, wcet2=500
OPERATION S1;S2
...
END

MACHINE Par
VARIABLES wcet1,wcet2
INITIALIZATION wcet1=1000, wcet2=500
OPERATION S1 || S2
...
END
```

In either case, we know that one part of the operation will take 1000 and the other 500 units of time to execute. Still we do not know how long it will take for the entire composite operation to execute. If, however, we assign class *transferred* to states `wcet1` and `wcet2` and introduce a minimization rule for both operators stating that:

$$\texttt{wcet}_{seq} = \texttt{wcet}_1 + \texttt{wcet}_2 + t_{comm}$$

$$\texttt{wcet}_{par} = max(\texttt{wcet}_1, \texttt{wcet}_2) + t_{sync}$$

After performing minimization, the resulting machines are (if $t_{comm} = 10$ and $t_{sync} = 20$):

```
MACHINE Seq
VARIABLES wcet
INITIALIZATION wcet=1510
OPERATION S1;S2
...
```

```
END

MACHINE Par
VARIABLES wcet
INITIALIZATION wcet=1020
OPERATION S1 || S2
...
END
```

Minimization has reduced the number of states in both cases and provided us with a more useful information on worst case execution time.

4.4 Machine Instantiation, Operator Priority and Properties

Service can be used in a formula describing composition in two ways: using its name (`serviceName` attribute of the `contract` element, or clause `MACHINE serviceName`), or by creating an instance of a certain service name. In the former case, all occurrences of *serviceName* in a formula refer to the same entity, that is, it is not possible to write something like *serviceName* $\|$ *serviceName*, since the same instance of one service cannot be executed in parallel with itself. Here we assume that, although services are stateless themselves, they can interact with certain persistent underlying resources (e.g., database) and it makes a difference whether the same service instance has been invoked twice, or two instances have been invoked once. We provide support for handling state and discuss it further in the Chapter 7. In the latter case, an independent instance of a service is created using `EXTENDS` clause: *A* `EXTENDS` *serviceA* means that literal *A* is created which is unique in a formula and represents an instance of machine *serviceA*. This instance is independent of any other instance of the same machine (e.g., can be executed in parallel with them). In the following example it is assumed that *serviceA* and *serviceB* are two services. First, service names are used to define composition:

$$(serviceA \rhd serviceB) \rhd serviceA$$

Here sequential composition of *serviceA* and *serviceB* is executed in sequence with *serviceA*. In both cases it is the same instance of *serviceA* meaning that rightmost literal *serviceA* knows the context in which leftmost literal *serviceA* has executed. For example, if the left literal changed something in a database or a file system without committing the changes, the

right literal will be aware of those changes. On the other hand, the same composition can be specified in the following way:

$$A_1, A_2 \texttt{ EXTENDS } serviceA$$
$$(A_1 \rhd serviceB) \rhd A_2$$

The difference is subtle, yet exists: services A_1 and A_2 are independent instances of the same service. They are best regarded as two independent state machines, with the same substitutions performed on the same state variables. In fact, A_1 and A_2 can be obtained from *serviceA* by renaming state variables and operations. This case is equivalent to the current state management that Web Services architecture offer: two invocations of a same service are unaware of each other, independent and without state transfer. As already mentioned, we provide support for state management (see Chapter 7), and that is the reason service instantiation mechanism is introduced: to be able to distinguish between literals representing invocation of the same service and independent instances of the same service. Bearing this in mind, it is possible to define the following reductions:

$$A \parallel A = A$$
$$A \parallel_P A = A$$
$$A \Box A = A$$

If we want to specify two instances of the same service executing in parallel we use:

$$A_1, A_2 \texttt{ EXTENDS } A$$
$$A_1 \parallel A_2$$

Since composition patterns are specified in infix notation, operator priority is important and defined as shown in Figure 4.6. The loop operator has the highest priority (unary precedes binary), then the sequence operator, followed by selection and both parallel operators. Choice operator has the lowest priority. Brackets can be used for other operand grouping if needed.

Now, we discuss commutativity, associativity and distributivity of composition operators. Binary loop is not commutative, that is:

$$W(e) \circlearrowleft_{P(e)} A \neq A \circlearrowleft_{P(e)} W(e)$$

The reason is obvious, since substitution that W effects upon loop exit variable e is not the same as the substitution that service A does. In fact, by

priority	operator
3	$\circlearrowleft_{P(e)}, W(e)\ \circlearrowleft_{P(e)}$
2	$\triangleright$
1	$\odot$, $\|$, $\|_{P(c)}$
0	$\square$

Figure 4.6: Operator Priority

definition, although in scope of service A, variable e must be independent from A's operations.

Sequence operator is also not commutative:

$$A \triangleright B \neq B \triangleright A$$

Proving this is also easy, since it is clear from definition that if left and right operands are switched, input and output messages of the composed service change.

Parallel composition without communication is commutative: it does not matter in which order concurrent services are specified since they execute independently:

$$A \parallel B = B \parallel A$$

Parallel composition with communication, however, is not commutative. The order of the operands is important since in case $P(c)$ holds, the results of the left operand will be used, and otherwise results of the right:

$$A \parallel_{P(c)} B \neq B \parallel_{P(c)} A$$

Similarly, selection operator is not commutative for the same reasons:

$$A \odot_{C(x)} B \neq B \odot_{C(x)} A$$

Finally, choice operator is commutative, since it is not known in advance which of the operands will provide results. Being non-deterministic in nature, switching operands for this operator will not produce any difference:

$$A \square B = B \square A$$

For the non-commutative operators of the same priority, expressions are evaluated from left to right, when no brackets are used for grouping:

$$A \parallel_P B \parallel_Q C \parallel_R D = (((A \parallel_P B) \parallel_Q C) \parallel_R D)$$

Binary loop operator is not associative:

$$A \circlearrowleft_P (B \circlearrowleft_Q C) \neq (A \circlearrowleft_P B) \circlearrowleft_Q C$$

Sequence operator, however is associative:

$$A \rhd (B \rhd C) = (A \rhd B) \rhd C$$

This is true due to the fact that conjunction is commutative and associative and therefore it does not matter in which order composite pre-condition is built. As for substitutions constituting operation body, they will always be executed in the same order (A $\longrightarrow$ B $\longrightarrow$ C), regardless of whether we first compose A with B, or B with C.

Parallel composition without communication is associative:

$$A \parallel (B \parallel C) = (A \parallel B) \parallel C$$

Since A, B and C are independent, the order in which parallel composition is specified is irrelevant. The resulting composition will execute all three services concurrently.

Parallel composition with communication, on the other hand, is not associative:

$$A \parallel_P (B \parallel_Q C) \neq (A \parallel_P B) \parallel_Q C$$

In the first case, choice is made between A and $B \parallel_Q C$ based on the predicate P. In the second case, choice is made between C and $A \parallel_P B$ based on the predicate Q which is not equivalent. The same holds for the selection operator:

$$A \odot_{P(x)} (B \odot_{Q(y)} C) \neq (A \odot_{P(x)} B) \odot_{Q(y)} C$$

Choice operator, being non-deterministic, is associative:

$$A \Box (B \Box C) = (A \Box B) \Box C$$

The order of composition does not carry any useful information here. In any case we still know only that one of the services A, B or C will produce the result of this composition. Knowledge whether B is first composed with C or A with B cannot determine the result in any way.

Commutativity and associativity properties are summarized in Figure 4.7.

Distributivity of operators will be examined next. Looping is distributive with respect to parallel composition without communication:

$$A \circlearrowleft_P (B \parallel C) = (A \circlearrowleft_P B) \parallel (A \circlearrowleft_P C)$$

	$\circlearrowleft_P$	$\triangleright$	$\parallel$	$\parallel_P$	$\odot_{C(x)}$	$\square$
commutative			$\surd$			$\surd$
associative		$\surd$	$\surd$			$\surd$

Figure 4.7: Operator Properties

This is true only if looping is synchronized, that is, service A evaluates predicate P for controlling loops of B and C at the same time. Since service A actually executes only once, this is not difficult to achieve implementing synchronous invocation, where A blocks until it receives response from both B and C, after which it continues calculating P for the next iteration.

Looping is distributive with respect to parallel composition with communication, under the same assumptions. In every iteration controlled by P, predicate Q is evaluated and either output of B or C is chosen:

$$A \circlearrowleft_P (B \parallel_Q C) = (A \circlearrowleft_P B) \parallel_Q (A \circlearrowleft_P C)$$

Looping is also distributive with respect to choice and selection, for the same reasons:

$$A \circlearrowleft_P (B \square C) = (A \circlearrowleft_P B) \square (A \circlearrowleft_P C)$$
$$A \circlearrowleft_P (B \odot_Q C) = (A \circlearrowleft_P B) \odot_Q (A \circlearrowleft_P C)$$

Sequence is distributive with respect to both parallel compositions and selection:

$$A \triangleright (B \parallel C) = (A \triangleright B) \parallel (A \triangleright C)$$
$$A \triangleright (B \parallel_P C) = (A \triangleright B) \parallel_P (A \triangleright C)$$
$$A \triangleright (B \odot_Q C) = (A \triangleright B) \odot_Q (A \triangleright C)$$

Sequence also distributes through choice:

$$A \triangleright (B \square C) = (A \triangleright B) \square (A \triangleright C)$$

Both parallel operators and selection distribute through themselves and choice:

$$A \parallel (B \parallel C) = (A \parallel B) \parallel (A \parallel C)$$
$$A \parallel (B \square C) = (A \parallel B) \square (A \parallel C)$$
$$A \odot_P (B \odot_Q C) = (A \odot_P B) \odot_Q (A \odot_P C)$$

$$A \odot_P (B \Box C) = (A \odot_P B) \Box (A \odot_P C)$$

Choice operator distributes only through itself:

$$A \Box (B \Box C) = (A \Box B) \Box (A \Box C)$$

This is possible because choice operator is non-deterministic and non-probabilistic. Only the possible reductions are important. For example, possible outcomes on the left side are either A or B or C. On the right side possible outcomes are (A or B) or (A or C). After reduction it is clear that three distinct possibilities are again A or B or C, although outcome A is more likely. However, that is implicitly also the case on the left side, since formulas are evaluated from left to right. Results for distributivity are summarized in Figure 4.8.

distributive	$\circlearrowleft_P$	$\rhd$	$\parallel$	$\parallel_P$	$\Box$	$\odot_C$
$\circlearrowleft_P$			√	√	√	√
$\rhd$			√	√	√	√
$\parallel$			√	√	√	√
$\parallel_P$			√	√	√	√
$\Box$					√	
$\odot_C$			√	√	√	√

Figure 4.8: Operator Distributivity

4.5 Verification of Composition Correctness

After establishing composition patterns, mechanisms for verifying composition correctness are introduced. A composition is correct if and only if it:

- type checks
- preserves the composite invariant
- terminates correctly

In the following sections each requirement will be examined and it will be shown how to determine whether an arbitrary composition created with composition patterns is correct or not with respect to this definition.

4.5.1 Type Checking

An abstract machine consists of pairs, such as (formal parameters, constraints), (constants, properties), and (variables, invariants). The first element of each pair defines literals that are used throughout the machine, while the second element defines relations that hold among the literals. There are two more entities characterizing a machine: initialization and operation. Type checking is used to assure that an abstract machine is consistent within the mentioned pairs and entities with respect to definition of literal type. Literals themselves can be scalar variables, sets, expressions or predicates. Type checking will be defined first, and then it will be shown how to perform it for an abstract machine.

Suppose that E is an expression and s is a set. Let $E \in s$ and let t be a set such that $s \subseteq t$. From this follows $E \in t$, and if t can be included in a larger set u, than $E \in u$ would also hold. The purpose of type checking is to establish that within an abstract machine there is an upper limit for such set containments. This upper limit, for the presented example, is the super-set of s and the type of E. The task is therefore to check whether sets and literals in an abstract machine are defined in such a way that type (as defined above) can be determined for all necessary machine elements. In other words, set containment within a machine must be finite.

Type of a predicate P is determined and denoted in a following way:

$$\texttt{ENV} \vdash \texttt{check}(P)$$

This means that within the environment `ENV` the predicate P type-checks. The environment is a finite collection of predicates such that for each free variable x in P, there is a predicate of the form $x \in s$ in `ENV`. Every predicate has a closed form, having no free variables. The question is what is the initial environment then? It could be empty, or if a generic statement is being proved, e.g., concerning a set a, then it is obvious that a has to participate in the type-checking, but it is not known what is its super-set. Set a itself is then used and denoted as $\texttt{given}(a)$. This has ground in the practice that such sets are usually introduced in informal specification with "Given a set a,...".

The function `check` evaluates either into checking of another (transformed) *predicate* `check(predicate)` or into equality of *types* `type1` $\equiv$ `type2`, which terminates type checking. Type can be either elementary (`type(`*expression*`)`), a superset of a given set (`super(`*set*`)`), a Cartesian product of two types (`type1` $\times$ `type2`), a power set of a type ($\mathbb{P}$(type)), or an identifier. We give an example:

$$given(\mathbb{N}, interval = [0..10), a \in interval, b = a \cdot a) \vdash$$
$$type(a) = interval$$
$$type(b) = super(a) = \mathbb{N}$$

If the set of natural numbers $\mathbb{N}$ was not defined however, the type checking would fail, since the type of b could not be determined. Some elementary transformations for calculating `check` are given in appendix B.4, while all can be found in [2].

Referring to the abstract machine from Section 3.4.1, type checking consists of the following requirements:

- Machine formal parameters (X, x), given sets S and $T = \{a, b\}$, constants c and state variables v have to be distinct
- Operation names in O have to be distinct
- S, T, a, b, c and v have to be non-free in constraints C
- v, x and X have to be non-free in properties P
- $\texttt{given}(X), \texttt{given}(S), \texttt{given}(T), a \in T, b \in T \vdash \texttt{check}(\forall x \cdot (C \implies \forall c \cdot (P \implies \forall v \cdot (I \wedge J \implies U \wedge O))))$

The last item requires an explanation. Using rules from appendix B.4 the last requirement can be decomposed as follows:

$$\texttt{check}(\forall x \cdot C)$$
$$\texttt{check}(\forall c \cdot P)$$
$$\texttt{check}(\forall v \cdot (I \wedge J))$$
$$\texttt{check}(U)$$
$$\texttt{check}(O)$$

This decomposition means that first universally quantified scalar parameters and their constraints are checked, then universally quantified constants and their properties, then universally quantified variables and their invariant, and finally initialization and operations. It remains to be seen how to type-check operation body.

An operation O is considered with one input parameter w and one output parameter u. Operation pre-condition is P and postcondition S:

$$\texttt{OPERATION}\ u \longleftarrow O(w)$$

$$\texttt{PRE}\ P\ \texttt{THEN}\ S\ \texttt{END}$$

Input parameter must have a type defined in the pre-condition clause. Therefore main substitution S is checked under universal quantification involving pre-condition P. If u, O and w are distinct and non-free in ENV, then we have to check:

$$\texttt{ENV} \vdash \texttt{check}(\forall w \cdot (P \Longrightarrow S))$$

which is equivalent to type checking operation body:

$$\texttt{ENV} \vdash \texttt{check}(u \longleftarrow O(w)P|S)$$

It is assumed here that the right side of the substitution that assigns value to output variable u is well typed.

4.5.2 Invariant Preservation

After type checking has been performed, the resulting machine must be proved not to break the composite invariant. The purpose is to establish the following:

- Composite initialization U must establish composite invariant I
- Composite assertion J must be deducible from composite properties P and invariant I
- Composite operation (with pre-conditions Q and operation body V) must establish composite invariant I

We list these requirements formally. Assuming constraints C and properties P it has to be proved that initialization U does not violate invariant I:

$$C \wedge P \Longrightarrow [U]I$$

Assuming constraints C, properties P and invariant I, it has to be proved that assertion J holds:

$$C \wedge P \wedge I \Longrightarrow J$$

Finally, assuming constraints C, properties P, invariant I, assertion J and pre-condition Q, it has to be proved that operation V does not violate invariant I:

$$C \wedge P \wedge I \wedge J \wedge Q \Longrightarrow [V]I$$

For these proofs the following is assumed:

- Machine formal parameters exist that satisfy their constraints
- Machine constants exist that satisfy their properties
- Machine input parameters exist that satisfy their pre-conditions

This, of course, has to be checked prior to the proving. After that C, P, and Q can be assumed and included on the left sides of proof obligation equations.

4.5.3 Correct Termination

After performing type checking and invariant preservation proofs, it has to be determined whether an abstract machine will terminate correctly and whether it is feasible.

Correct termination deals with establishing a post-condition. Let us go back to the simple pre-conditioned substitution:

$$[P|S]R \iff P \wedge [S]R$$

The question is, what happens when pre-condition P does not hold. Since P on the right side of the equation is false, the entire right side is also false, regardless of R. In this case substitution $P|S$ is not guaranteed to establish anything (R included). Such a substitution that cannot establish anything (is not guaranteed to establish anything) is *non-terminating* substitution. We now explore termination of substitutions in more details.

Given a substitution S, expression $\texttt{trm}(S)$ denotes a predicate that holds when substitution S terminates. It is rather difficult to define "termination" directly, therefore, we reason about its negation $\texttt{abt}(S)$. An aborting substitution is a substitution that cannot establish any post-condition. Such substitution will likely cause a deadlock if used in a composition. Therefore, it is essential that we are able to detect it. For given substitution S and any post-condition R, $\texttt{abt}(S)$ is defined:

$$\texttt{abt}(S) \iff \neg[S]R$$

and subsequently, correct termination is defined:

$$\texttt{trm}(S) \iff \neg\texttt{abt}(S)$$

Let us prove that definition of aborting substitution can be simplified:

$$\texttt{abt}(S) \iff \neg[S](x = x)$$

Proving implication from left to right is easy. Assuming $\texttt{abt}(S)$, that is, assuming $\neg[S]R$ it obviously follows that $\neg[S](x = x)$. Proving implication from right to left is, however, a bit more difficult. Assume $\neg[S](x = x)$. It is obvious that $\forall x \cdot (R \implies x = x)$ for any R. It can be shown that establishment of a post-condition is monotonic, that is, for a substitution S, variable x and predicates A and B the following holds [2][1]:

$$\forall x \cdot (A \implies B) \implies ([S]A \implies [S]B)$$

It means that $[S]R \implies [S](x = x)$ for any R, and by contraposition $\neg[S](x = x) \implies \neg[S]R$ for any R. Both directions of implication being proven, equivalence $\texttt{abt}(S) \iff \neg[S](x = x)$ has also been proved. The next result directly follows:

$$\texttt{trm}(S) \iff [S](x = x)$$

The last result was required to determine whether generalized substitutions terminate. For given pre-condition P, substitutions S and T and variable z, the following holds:

$$trm(P|S) \iff P \wedge trm(S)$$

$$trm(S \square T) \iff trm(S) \wedge trm(T)$$

$$trm(P \Rightarrow S) \iff P \Rightarrow trm(S)$$

$$trm(S||T) \iff trm(S) \wedge trm(T)$$

$$trm(@z.S) \iff \forall z \cdot trm(S)$$

Loop (while substitution) is omitted here, and for a good reason: in general case as we defined it, judging correct termination of a loop is extremely difficult. Therefore, certain restrictions are now introduced that will enable determining correct termination of a loop. Two conditions must be met for a loop to terminate [130]:

- The loop must make progress toward establishing the termination condition described by the loop guard under any initial conditions allowed by pre-condition.
- The loop guard must be strong enough to force termination after a finite number of iterations.

Termination of loops is generally based on investigating conditions on a suitable variant function [6]. Therefore, the loop is rewritten as follows:

$$\texttt{WHILE}\ P\ \texttt{DO}\ S$$

[1] Chapter 6.2, pp 287-288, property 6.2.2

`INVARIANT` I `VARIANT` V `END`

Let us assume that substitution S works with variable x. Termination of such a loop is equivalent to:

$$I$$

$$\forall x \cdot (I \wedge P \Longrightarrow [S]I)$$

$$\forall x \cdot (I \Longrightarrow V \in N)$$

$$\forall x \cdot (I \wedge P \Longrightarrow [n := V][S](V < n))$$

First two lines indicate that invariant has to be true and that loop body must reestablish the invariant. The third line defines that variant part of the loop evaluates to set of natural numbers N. Finally, if n is a variable that is assigned variant in each iteration ($n := V$), after loop body is executed (S), variant must decrease ($V < n$). That way it is ensured that loop makes progress towards termination. In case a more complex variant is required (e.g., comparing characters instead of a simple integer counter), it can be easily generalized assuming that a set in which it evaluates is ordered by some relation.

We now deal with feasibility. Let us observe a guarded substitution, where P is a predicate, S is a substitution and R is post-condition:

$$[P \Longrightarrow S]R \Longleftrightarrow (P \Longrightarrow [S]R)$$

If P does not hold, because of the implication, this substitution is able to establish any post condition R. Such substitution is called non-feasible. Similar to the terminating substitution, feasible substitution is defined: $\texttt{fis}(S) \Longleftrightarrow \neg[S](x \neq x)$, that is, feasible substitution is negation of non-feasible, which in turn can establish even $(x \neq x)$. Following are rules for calculating feasibility of generalized substitutions:

$$fis(P|S) \Longleftrightarrow P \Rightarrow fis(S)$$

$$fis(P \Box T) \Longleftrightarrow fis(S) \vee fis(T)$$

$$fis(P \Rightarrow S) \Longleftrightarrow P \wedge fis(S)$$

$$fis(S||T) \Longleftrightarrow fis(S) \wedge fis(T)$$

$$fis(@z.S) \Longleftrightarrow \exists z fis(S)$$

4.6 Composable Architecture

It is very difficult to solve the problem of service composability without defining an environment in which the composition takes place. This is the reason why an abstract composable service architecture is introduced. It is also impossible to provide a meaningful interface specification of an open component, without considering the context of use of the component in a particular environment [81]. Therefore, composability is attributed to an architecture [142]. The architecture is defined as tuple $A(E, O)$ where E is a set of initial (atomic) services, and O is a set of composition operators. An operator $o \in O$ is a function that maps two (or more) services to a new service: $o : E \times E \times ...E \rightarrow C$, where C is a set of complex (composed) services and $E \subset C$. Set E has finite number of elements, while set C has inifite elements. Naturally, it possible to "recompose" already composed services, hence a more complete definition of a composition operator is: $o : E \times E \times ...C \times C \rightarrow C$.

However, not all composed services are valid members of the architecture. Therefore a function $correct : E \times E \times ...C \times O \rightarrow \{true, false\}$ is introduced. Function $correct(e_1, ..., e_n, c_1, ..., c_m, o)$, where $e_1...e_n \in E$ and $c_1, ..., c_m \in C$ and $o \in O$, returns *true* if the composition of elements $e_1, ..., e_n, c_1...c_m$ using composition operator o is correct, and returns *false* otherwise. The function *correct* is calculated for the composite element e in the following way:

$$correct(e) \Longleftrightarrow check(e) \wedge proof(e) \wedge trm(e) \wedge fis(e)$$

where $proof(e)$ denotes invariant preservation.

The idea of composable service architecture is that if *forbids* composition of incorrect services, where correctness is defined by the function *correct*. Note that it is assumed that constituent services are themselves correct, and reasoning is performed about the correctness of the composed service only. In the case where constituent services are incorrect themselves (they do not satisfy type checking, proof obligation, termination or feasibility) correctness of the composite service cannot be guaranteed. However, since we require formal service description there is a potential for verification of service implementation with respect to specification, as much work has already been done in this area [102].

The point where our approach to composability differs from others is in the definition of the composable architecture. It is an architecture that supports composability with respect to a property (in our case correctness). The traditional approach to composability works in the domain of elements being composed. We transpose this into the domain of system architecture

which must guarantee *safety* property. The safety property must be possessed by all elements of the architecture. An architecture is then composable with respect to a safety property if and only if it allows only the composition of elements having this property. That way the focus is shifted from design of systems to design of architecture. Once a composable architecture has been defined, systems can be composed out of valid elements with safety guarantees.

Complete composition and verification example is demonstrated in Appendix D.

4.7 Trust, Optimizations and Reputation Systems

One important issue that needs to be explained is the type of correctness that composable service architecture guarantees. In the area of proving program correctness, two general types of correctness proofs can be distinguished [59]:

- Type-1 correctness: proof of existence of correct behavior
- Type-2 correctness: proof of the non-existence of incorrect behavior

If we adopt these definitions in our scenario, we would ideally like to provide type-2 correctness. When proving program correctness, this type is generally considered to be impossible to achieve. Function *correct* guarantees type-1 correctness. However, how strong are its type-2 correctness guarantees? To be able to answer to this question, we introduce a notion of *trust* in composable service architecture. Trust can be modeled at three different levels:

- single services
- composition patterns and composite services
- reputation systems

Solving trust at the level of a single service is relatively straightforward: when a service is deployed to a directory, its contract is verified for correctness. Publication of incorrect services is not allowed. After verification is passed (function *correct* evaluates to true), it is assumed that a service is correct and subsequent verification is performed only when its contract changes. Therefore, all services residing in a directory are considered correct.

For discussing trust at the level of composition patterns and composite services, assume that we have two (or more) abstract machines M_1 and M_2 representing Web Services and having distinct state variables x_1 and x_2 and invariants I_1 and I_2. Let $P_1|S_1$ and $P_2|S_2$ be two Web methods (operations) of these services. If any composition pattern is applied to M_1 and M_2, such that:

$$correct(M_1 \ \texttt{op} \ M_2) = true$$

where $\texttt{op} \in \{\rhd, \|, \|_p, \odot_C, \Box, \circlearrowleft, \circlearrowleft_q\}$, the problem of trust is solved. Essentially, correctness is verified after each composition, without taking into any account correctness of operand services. However, suppose that both operand services are correct:

$$\forall x_1 \cdot (I_1 \wedge P_1 \Longrightarrow [S_1]I_1)$$

$$\forall x_2 \cdot (I_2 \wedge P_2 \Longrightarrow [S_2]I_2)$$

Since it was assumed that x_1 and x_2 are independent, these equations can be rewritten:

$$\forall (x_1, x_2) \cdot (I_1 \wedge I_2 \wedge P_1 \wedge P_2 \Longrightarrow [S_1]I_1 \wedge [S_2]I_2)$$

It can be shown [2][2] that the following holds for substitutions S and T and post-conditions P and Q, if S and T work on distinct (independent) state variables x and y, and x is non-free in Q and y is non-free in P:

$$[S]P \wedge [T]Q \Longrightarrow [S \parallel T](P \wedge Q)$$

Since x_1 and x_2 are distinct and independent, and x_1 is non-free in I_2 (does not appear in I_2) and x_2 is non-free in I_1 (also does not appear in I_1), applying this to $[S_1]I_1 \wedge [S_2]I_2$ we have:

$$[S_1]I_1 \wedge [S_2]I_2 \Longleftrightarrow [S_1 \parallel S_2](I_1 \wedge I_2)$$

which leads to the main result:

$$\forall (x_1, x_2) \cdot (I_1 \wedge I_2 \wedge P_1 \wedge P_2 \Longrightarrow [S_1 \parallel S_2](I_1 \wedge I_2))$$

Rewriting the last equation like:

$$\forall (x) \cdot (I \wedge P \Longrightarrow [S]I)$$

where $x = \{x_1, x_2\}$, $I = I_1 \wedge I_2$, $P = P_1 \wedge P_2$ and $S = S_1 \parallel S_2$. This means that operation specified as $(P_1 \wedge P_2)|(S_1 \parallel S_2)$ preserves the invariant $I_1 \wedge I_2$. It

[2]Chapter 7.1.3, pp 311-312, theorem 7.1.1

can also be shown that any combination of operations using other generalized substitutions will also preserve invariant $I_1 \wedge I_2$. This suggests a mechanism by which correct abstract machine can be composed out of correct machines M_1 and M_2: the new machine invariant and pre-conditions are constructed by conjuncting composed machines' invariants and pre-conditions, while substitution (operation) is achieved by multiple substitution, using adequate rules for composition pattern. In this case, verification of the composed machine is not necessary. It is enough that we know if starting machines are correct. Therefore, instead of proving complex invariant of the form $I_1 \wedge I_2 \wedge ... \wedge I_n$, process of verification can be optimized as it is enough to prove that starting services (operands) are correct only. However, if the operand services do not work on the independent variables, as is the case of sequential composition, the above claim does not hold since it cannot be assumed that invariant will be reestablished. In that case composite invariant must be proved.

The discussion above is valid assuming that contract is a faithful representation of the underlying service. We implicitly assumed that every service behaves strictly according to its contract and proved correctness of *contract* composition accordingly. We did not address the issue of implementation correctness. That is justified by one of the basic premises of service-oriented computing: we do not have the access to service implementation, but only to service description. The important question is how far a service contract can be trusted? There are at least two situations where contract does not represent underlying service accurately:

- Mistake/inexperience of service developer/deployer causing publication of correct but otherwise misleading (inaccurate) contract.
- Intentional and malicious contract forgery causing publication of malicious and dangerous content under false contract.

Although some initial steps have been taken towards proving correctness of implementation with respect to formal specification [102], which would at least partially solve this issue, both cases are still virtually untraceable and undetectable. Therefore, in order for discussion above to be complete, a reputation system is required to maintain robustness and decrease probability of false/misleading/erroneous content being advertised through otherwise well formed service contract.

The main role of a reputation system is to collect, distribute and aggregate feedback about services' past behavior [140]. It should help to facilitate decision on whom to trust, encourage trustworthy behavior and deter unskilled and dishonest parties. Introducing a reputation system to the formal environment such as composable service architecture poses certain problems,

namely inability to quantify or describe how the system will behave in presence of errors/frauds and how fast (if ever) it will converge towards a stable state where only accurate and honest contracts and services will be utilized. Despite that, relative success of the existing reputation systems and our obvious inability to guarantee that every contract is honest necessitate deeper investigation of reputation systems.

Reputation systems have their origins in online auction transactions. Although online auctions (e.g., eBay) offer almost perfect environment for fraud and deceit, their reputation systems keep the rate of successful transactions at surprisingly high percent [80, 186]. After a transaction in such a system is complete, both buyer and seller have the opportunity to rate each other using several measures (e.g., friendliness, response, quality of the product). When designing a reputation system, the following issues have to be addressed: eliciting, distributing and aggregating feedback. Eliciting feedback ensures that feedback is provided at all, that a negative feedback can also be easy elicited (this is a problem sometime due to fear of retaliation), and that feedback is honest. Distribution of the feedback ensures that feedback cannot be invalidated by name (identity) change and that feedback results can be shared among different systems. Finally, feedback aggregation should ensure that meaningful results are produced that can help parties in selecting trusted cooperation partners.

Having these guidelines in mind a reputation system is introduced that keeps track of submitted requests and responses and ranks performed operations according to several criteria:

- Did operation functionally succeed? Did return parameters match contract?
- Did operation succeed non-functionally? Were non-functional post-conditions established (e.g., did operation last less than stated worst case execution time)?
- Were there exceptions thrown in any scope up to the scope of the caller? [3]
- Did operation terminate (were there deadlocks)?
- Is operation feasible (does it produce uniform results)?

Determining answers to these questions is mandatory for every operation. The ranking is then maintained for single as well as for composite services.

[3] Exceptions and scopes are defined in Chapter 7

When a single service ranks low, it is an indication that either its contract is malformed, or that it has an implementation error. When a composite service ranks low, reasons can be either on the side of deployer of one or more constituent services (same reasons as for single services), or on the side of service composer and consumer (specifying correct composition, but one that does not match problem specification). Determining and enforcing liability in such cases is the task of the topmost layer of a service-oriented architecture (managed services, see Introduction), and we will not address this problem further here. Although reputation system is always maintained, its usage is not mandatory. Composition can be performed and executed without consulting ranking of contributing services. Having such a system, however, improves trustworthiness and helps in eliminating (to some extent) negative and unwanted behavior.

Chapter 5

Composing Web Service Design Patterns

Using developed composition operators and verification rules, basic Web Service design patterns are described in this chapter: proxy pattern, facade pattern, security patterns, dynamic input pattern, logger pattern, load balancer pattern, publish-subscribe pattern and producer-consumer pattern. These patterns can be regarded as coarse-grained building blocks for composite applications design. This approach gives the ability to formally introduce and verify application of design patterns in complex service-oriented software systems.

5.1 Service Design Patterns

Design pattern is description of the core of an engineering problem that occurs over and over again in practice, and description of the solution to that problem, such that this solution is reusable in different contexts. It can also be said that design pattern is a best practice solution to a common recurring problem. Design patterns were first described in [7] and applied to civil engineering. In terms of software engineering, design patterns were introduced in [53], specifically targeted to object-oriented systems. Each pattern has four elements: name, description of a problem it tries to solve, elements of a solution, and consequences (results and trade-offs). The first element deals with naming, which is important in communicating design elements, while the other three elements perform analysis.

The main benefits of design patterns in software engineering are:

- Design patterns provide high-level language for describing design issues.

- Design patterns provide much of the design work upfront.
- Combinations of design patterns lead to development of reusable architectures.

After this pioneering work, many design patterns have been identified for object-oriented software systems, and some of them have become standard elements of many design projects (e.g., Observer, Facade, Command) or have even been incorporated in several programming languages (e.g., Factory Method).

With the introduction of new paradigms and technologies, such as service-oriented computing, it has been noted [131] that usability of "new" patterns seems to be approaching zero. One reason could be that all major patterns have already been discovered, but the true reason is rather that we are thinking at the new level of granularity when designing service-oriented applications. Instead of designing at the object/class level, we work at the subsystem/application level. Therefore, existing design patterns are not applicable and/or adequate anymore. There is a need to develop design patterns for service-oriented architecures.

Currently, surprisingly little research is being done in the area of developing methodologies of "good" service-oriented engineering, with the notable exception of [155] and [17, 18], where several practical Web service design patterns are explained. The design patterns proposed in this chapter should be considered complementary to them, as "best-effort" [168] solution for enforcing quality design.

There are two main differences between object-oriented and service-oriented design patterns:

- different levels of design granularity and abstraction
- service-oriented design is not inherently client-server, but dynamic: many clients can choose among many servers (services), resulting in a community of peers that invoke each others

In the following sections, several Web Service design patterns will be identified. Instead of describing them using UML diagrams (similar to OO patterns), we will use composition operators to formally express them. The key difference will thus be the ability to formally introduce and verify application of design patterns in complex service-oriented software systems.

5.2 Synchronous and Asynchronous Invocation

This is actually not a design pattern, but a design decision that precedes using other patterns. This choice will impact the selection of other patterns as well as level of coupling between composite services.

Synchronous invocation is in fact remote procedure invocation, wrapped in SOAP message. That means that parameters are passed to a defined remote method, and the caller blocks until a response from the called method is received, again as part of a SOAP message. On the server side, SOAP message is decoded and translated to back-end object method call. Therefore, it is very easy to use this kind of invocation, however it introduces tight coupling between client and server.

On the other hand, asynchronous invocation is based on exchange of XML documents and is not constrained to execution of predefined methods on the server side. The SOAP message does not map directly to a remote method, but rather requires additional processing in order to interpret it (possibly according to a predefined schema). Client does not have to block (although it may choose to do so anyway) while waiting for the results, but can continue its own processing until it receives XML document as a response. This kind of invocation is more difficult to implement, but enables loose coupling between the calling and the called service.

Many design patterns will try to leverage advantages of asynchronous (document-based) invocation against the simplicity of synchronous (RPC-based) invocation. In the following patterns, a distinction between synchronous and asynchronous invocation will be made only where relevant, otherwise it will be assumed that both types can be used equally.

5.3 Proxy Pattern

The easiest way to access a Web Service is to do so directly, using some specific API on the client side that connects to the WSDL interface of a target service. However, this method creates strong coupling between calling and called service, as well as makes reuse difficult, since the same calling code has to be repeated many times. A solution is to use a service proxy pattern that decouples called from the calling service by using surogate (proxy) service instead of a target service.

5.3.1 Single Proxy

Single proxy pattern is used to access single Web Service indirectly. It inserts additional proxy Web Service between client and server. The task of a proxy service is to read input parameters, invoke target Web Service and receive results. Therefore, communication with the target service is implemented only once, inside proxy service. This facilitates reuse and it is also possible to change the interface of the back-end service without notifying the clients, as it is enough to update only proxy accordingly. Given a *client* service, *target* service and its *proxy*, this pattern can be descibed with the following composition (Figure 5.1):

$$client \triangleright proxy \triangleright target$$

Figure 5.1: Proxy Pattern

The main benefit of using a proxy pattern is that it enforces loose coupling between calling and called service. Also, this extra layer of indirection can be used for logging or load balancing, as will be shown in the following patterns.

5.3.2 Multiple Proxy (Transformer)

A service can have more than one proxy. In case of multiple proxies, they are used to convert (transform) interface of the target service according to the expectations of different clients. Therefore, the alternative name for this pattern is transformer. Transformer can be used to enforce understanding on semantic meaning of parameters, or to help connecting services developed with different back-end technologies.

The fact is that just adopting XML and SOAP does not guarantee interoperability. The reason is that there are many flavors and implementations of SOAP and despite opposite claims, practice has shown that they are not compatible with each other. The issue of data types is extremely tedious to manage if partner services are exchanging other types than primitives like integers and strings. The problem is exacerbated by the fact that different service providers will provide different sophistication when publishing their services' descriptions and interfaces. The issues of naming data, assigning business meaning to XML data and serialization/deserialization are just some of the real-world problems. Since it is not realistic to expect that single

XML schema can be enforced to all business partners, the idea is to provide a separate proxy for every partner, resulting in multiple proxy (transformer) pattern. This of course requires that partner schemas are known in advance. The composition describing this pattern is (Figure 5.2):

$$(client_1||...||client_m) \rhd (proxy_1 \odot proxy_2 \odot ... \odot proxy_n) \rhd target$$

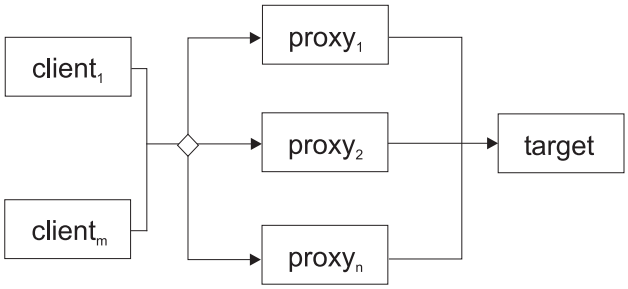

Figure 5.2: Multiple Proxy (Transformer) Pattern

This pattern should not be used to connect to multiple services using multiple proxies, as this is the task of the Facade pattern. Instead, multiple proxy should always connect to a single target service.

5.3.3 Proxy with Channel

Further expansion of the proxy pattern is decoupling of communication protocol and business interface. A channel service is introduced that deals with communication protocol issues (e.g., SOAP) while proxy service deals with parameters and business logic only. That way communication protocol can be changed without changing either calling service or service proxy. Multiple proxies can share a single channel, or choose among several available ones. The composition expressing this pattern is (Figure 5.3):

$$(client_1||client_m) \rhd ((proxy_1 \odot ... \odot proxy_n)$$
$$\rhd (channel_1 \odot ... \odot channel_p)) \rhd target$$

In this pattern, calling service sends a request to one of the proxies. Proxy performs data transformation and prepares the request for the target service. Then it selects appropriate channel and sends raw data. The channel packs received data into appropriate message and actually invokes the target

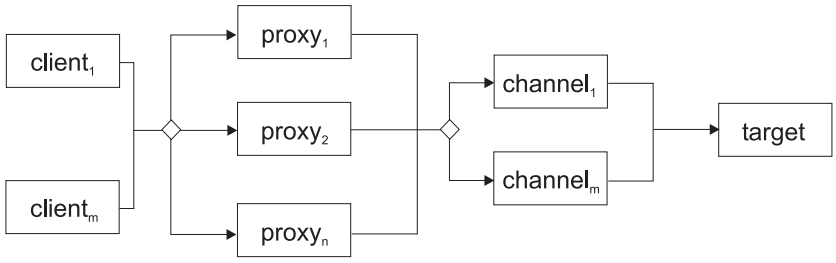

Figure 5.3: Proxy With Channel Pattern

service. The channel also receives results, and returns them to proxy for relevant formatting. This means that proxy and channel are strongly coupled, but the important issue is that channel can be changed without notifying the client service, meaning that client is now decoupled from business as well as from communication interface.

5.4 Façade Pattern

While proxy pattern facilitates access to a single Web Service, façade pattern performs the same for compositions of services. The problem that façade pattern tries to solve is how to access a composition of Web Services in an efficient and reasonable manner. Let us first try to identify design issues and problems that a client may face when trying to access a composition of services.

Contrary to the classic Façade pattern (used in object-oriented design), multiple network calls are not a problem when invoking compositions of Web Services, since request to execute a composition is sent to the composition server which manages and optimizes all network calls itself (see Chapter 7, Implementation). However, the problem is the coupling between the calling service and all called services. Although mediated by the composition server, the caller has to have intimate knowledge of all services involved and their interfaces. This makes replacement of services in composition difficult. Reusability is not that problematic, as composition can be stored in a directory and then re-invoked when necessary, but the problem of coupling remains. Finally, in this case it is up to the client and/or composition server to specify transactional and other non-functional behavior of the constituent services which is a weak design point as it can introduce potential inconsistencies.

The key idea is to represent fine grained operations offered by partner Web

Services in a composition through a single Façade service, which encapsulates all necessary calls and offers a coarse grained composite operation to the caller service. Actually, this is equivalent to having service provider deploying another Web Service that corresponds to the composition of several other services. The Façade pattern can be expressed (Figure 5.4):

$$(client_1||...||client_n) \rhd facade \rhd (target_1 \circ target_2... \circ target_m)$$

where $\circ \in \{\rhd, \odot, \square, ||, \circlearrowleft\}$.

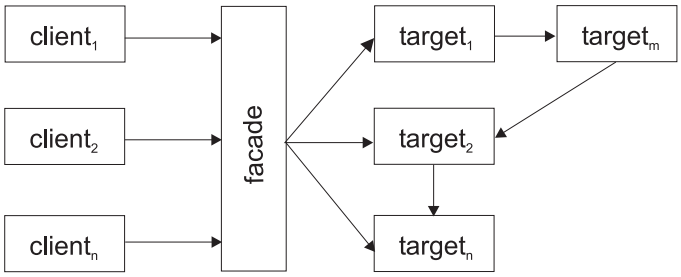

Figure 5.4: Façade Pattern

Façade and target services are strongly coupled here, but even that can be eliminated by applying proxy pattern for each target service. This is the question of the composition complexity, as combining proxy and façade patterns can easily prove to be an overkill for the entire application, depending on its size and requirements.

The benefit of applying façade pattern to Web Service composition is in decoupling caller service from the partner services comprising the composition. Also it is more natural that provider of one or more partner services specifies transactional and other non-functional behavior (e.g., security, timeliness) instead of caller service or composition server (which has to enforce those rules anyway). Finally, composition reusability is better if entire composite service is provided upfront, and not stored as a composition in a directory.

Next, a difference between a syncrhonous and asynchronous façade patterns will be discussed.

5.4.1 Synchronous Façade

If a synchronous façade pattern is used, caller service will block and wait until entire composition is executed. This type of invocation is used when

transactional attitude is strict (attribute `REQUIRED` is used) and client must know the result of entire composition execution before proceeding further.

A good example where synchronous façade has to be used is a classical bank transfer scenario. A composition has to be created that accepts two bank account numbers, security credential and an amount that is to be transfered from one account to the other. This can be solved by composition of three services: authorization service, withdrawal service and deposit service.

Caller service cannot continue its execution before all three partner services complete. Furthermore, if any of partner services abort, entire operation has to be aborted and compensated, which is the task of the façade service. Note again that transactional behavior can be solved at the level of composition server, but ensuring consistency by the service provider itself (assuming that all three partner services are published by the same provider, e.g., a bank) at the façade level is more natural and recommended. Finally, if the façade is running inside the same application server as partner services, network overhead will be also eliminated.

5.4.2 Asynchronous Façade

Asynchronous façade is used when the nature of a composition is such that a caller service does not require immediate response, furthermore, where such behavior would be harmfull to the overall business logic and performance. This case is usually dictated by scalability and timeliness requirements. Such compositions allow client services to continue their own processing while queuing their requests and performing batch processing. The task of the asynchronous façade is exactly that: to queue client service requests and respond after all composite services have finished. The client service can be re-invoked in the meantime or can continue its own processing.

The notable example of a composition where asynchronous façade is a good solution is airline reservation system. A client service invokes a composition of services that make hotel and flight reservation for a given date, and a payment service. If a synchronous façade pattern is used to encapsulate this composition, two problems are encountered: long delay and reliability. Checking flight and hotel availability can take a long time, and even involve human processing. Therefore, for business cases where transactions can take long time to execute, it is unacceptable for a client service to block and wait. Using asynchronous façade, a client is free to continue its own processing the moment it submits the request to the façade service and it will be notified once the composition terminates, successfully or unsuccessfully. Fault-tolerance is also a problem with long running business transactions. If a synchronous façade has been used and one partner service aborts (e.g., ho-

tel reservation), entire composition will abort. However, when asynchronous façade is used, the request will be kept in a façade queue and the execution will be retried, thus making long running transactions more resilitent and reliable.

5.5 Security Patterns

In this section it will be demonstrated how security patterns can be constructed and expressed using composition operators. There are two ways security requirements can be addressed in a composable service architecture. The first solution is to rely on the pre-condition/post-condition mechanism only, that is, to exploit inherent security capabilities of partner services that build the composition. Although easier, as it requires only careful specification of requirements in service contracts, this solution can result in incorrect composition in case when partner services do not support certain security properties themselves. The second option is to 'reinforce' composition with several dedicated services, creating a security pattern or wrapper for the entire composition.

Basic security attributes that can be reinforced in a composition are: transport, encryption, access control and message integrity. Assuming that dedicated services and channels supporting each of the attributes exist, one possible security pattern for simple proxy invocation would be (Figure 5.5):

$$client \triangleright encrypt \triangleright https \triangleright channel \triangleright decrypt \triangleright facade$$

$$\triangleright (authorize \square access \square integrity) \triangleright proxy$$

Transport protection deals with security of the channel that transmits data from client to facade/proxy. It encrypts the connection between two services, thus protecting the channel. Secure Sockets Layer (SSL) and HyperText Transfer Protocol Secure (HTTPS) can be used for this purpose. While in transit, all data is secure. However, data itself is not encrypted, meaning that at the endpoint (receiving point), information is easily read. Therefore, all data is also encrypted before being sent to the channel and decrypted at the receiving end. Finally, after being received by the facade and before being forwarded to proxy, authorization, access control and message integrity verification can be performed. It is up to the architect to decide which of these security services should be used in a particular use case, since they introduce significant overhead.

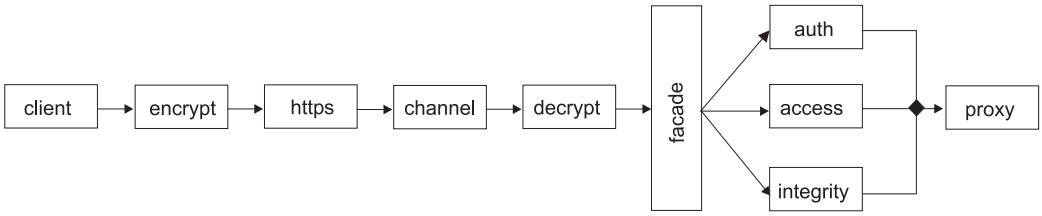

Figure 5.5: Security Pattern

5.6 Dynamic Input Pattern

Web Service messages can be very complex, comprising many input/output parameters. It is in accordance with basic postulates of service- oriented computing, which recommend using small number of operations with relatively large and complex messages. Furthermore, those messages are often not known until runtime, that is, they are constructed dynamically. Data necessary for message construction can come from an XML file or a relational database.

Apart from the problem of complex input messages, many Web Services operate with persistent resources which have to be identified (e.g., relational database, table name and primary key). This data is also not known during design time. In such cases it is convenient to remove logic for dynamic message construction outside of the client, into a combination of service proxy and configuration manager. This pattern can be expressed by (Figure 5.6):

$$client \rhd (dynamicProxy||configurator) \rhd integrator \rhd target$$

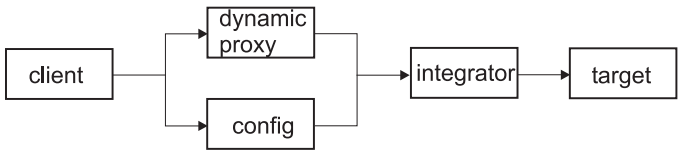

Figure 5.6: Dynamic Input Pattern

In this pattern, dynamic proxy performs the usual interface and data transformation (decoupling client from server) and configurator consults persistent resource to retreive necessary data. Both services forward their output to integrator which generates the complete request by filling missing information that it receives from the configurator. In order to remove latency

required for consulting persistent resource, configurator can cache configuration data in memory, accelerating dynamic message construction and lowering overall response time.

5.7 Logger Pattern

As already discussed, reputation systems are a neccessary part of service-oriented architecture. In order for reputation system to be fair and usable, it is convenient to introduce standard design pattern that requires all services to log their input and output messages. That way logging logic is removed from the partner services. Apart from being useful for building standardized reputation systems, logging pattern can be used for debugging and testing of service-oriented applications, which is an issue often neglected today. Designers and developers creating service-oriented applications today are more often than not in complete darkness when trying to debug and diagnose application errors. Even access to error logs is a serious problem since target services usually execute in different application containers and access to their logs is not always trivial and/or possible. Therefore, introducing a standard logging pattern for all services can help in the aspect of validation of complex service-oriented applications.

Practice has shown that it is reccomendable to perform logging in a database table and not in a plain file for several reasons, one being that XML messages are very long, resulting in long and unusable log files. Also, it is desireable to have related messages grouped together, which cannot be guaranteed because of the concurrency issues. If all related messages are stored in the database tables, a simple cross join can retrieve all relevant data. Hence both issues (search and relations) are solved. The logger pattern can be expressed (Figure 5.7):

$$client \rhd (proxy||requestLog) \rhd (target||responseLog)$$

Alternatively, proxy can also log response, eliminating the need for separate response logger:

$$client \rhd (proxy||requestLog||responseLog) \rhd target$$

5.8 Load Balancer Pattern

Frequently, a pool of Web Services that perform the same function is available. Instead of invoking them on a random basis, they can be used for load

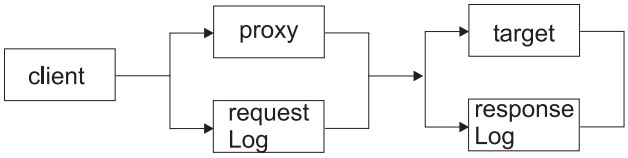

Figure 5.7: Logger Pattern

balancing using proxy pattern. Inserting a proxy between clients and a pool of target services, and equipping that proxy with a load balancing algorithm optimizes the performance of the whole system. The role of a load balancer is thus to distribute client requests among available service instances. For a given target service (functionality) T, load balancer pattern can be expressed (Figure 5.8:

$$TI_1, TI_2, ..., TI_m \texttt{ EXTENDS } T$$

$$(client_1||...||client_n) \rhd proxy \rhd loadBalancer \rhd (TI_1 \odot ... \odot TI_m)$$

The decision whether proxy and load balancer are implemented synchronously or asynchronously has significant impact upon the choice and performance of the load balancing algorithm. If synchronous load balancer is implemented, only a guessing algorithm can be used, since there is no way that load balancer can know for sure which services are available and which are not. This decision has to be taken based upon imperfect historical data. This is a push model, where requests are pushed to the target service instances without their cooperation. On the other hand, if communication between load balancer and target services is implemented asynchronously, a pull model can be used. Load balancer can store requests in a queue and target instances can retrieve and process them (pull) once they are free. The pull mechanism (or producer-consumer) is described in more detail in the Section 5.10.

5.9 Publish-Subscribe Pattern

The idea of introducing publish-subscribe mechanism into service-oriented middleware has been proposed in [67] to facilitate critical infrastructure protection operations. We will expand and formalize this approach. The main actors in this pattern are publisher services, subscriber services and communication middleware services (message bus). Publishers produce values (computations, measurements) and put them on the message bus. Subscribers opt

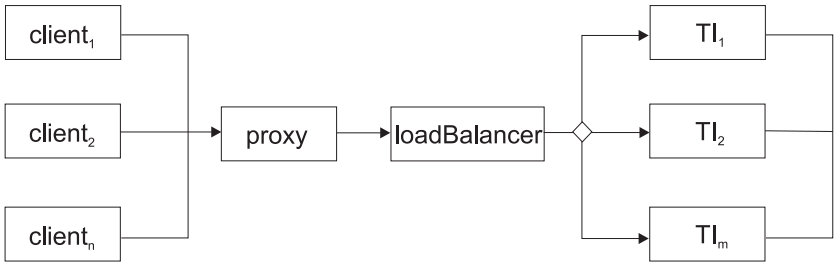

Figure 5.8: Load Balancer Pattern

to receive a selection of available data from the bus. The bus itself comprises aggregator, transformer, queue and directory services. Aggregators perform data merging, transformers perform transformation on raw data (e.g., a FFT can be performed for a given measurement), queue stores raw, aggregated or transformed data, while directory collects description of available data in the queue. Subscribers consult directory when choosing data to subscribe to.

The pattern can be described as follows (also in Figure 5.9):

$$\{[(target_1 \rhd publish_1)||...||(target_n \rhd publish_n)]\rhd$$
$$(aggregate_1 \Box ... \Box aggregate_p \Box transform_1 \Box ... \Box transform_q)\rhd$$
$$(queue||directory)\}||$$
$$\{(client_1 \rhd subscribe_1)||...||(client_m \rhd subscribe_m)\rhd$$
$$(queue||directory)\}$$

In the context of critical infrastructure (e.g., the power grid) protection, publishers are sensors that give various measurements along the grid elements (power plants, turbines, generators, transformators and distribution lines), subscribers are fault predictors that subscribe to information they require to perform prediction, while message bus is the heterogenous communication middleware that must fulfill non-functional requirements of the subscribers. Each subscriber can specify a set of constraints that must hold (e.g., a fault predictor may require timely and secure delivery of measured data).

The proposed pattern offers additional availability benefit: composability over multiple domains. Frequently, subscribers will require data that originate at different providers, separated by legal, safety and technological barriers. Flexibility of unified interfaces, correctness guarantees and composability enables aggregation of data that would otherwise be inaccessible in the classical monolithic application scenario. Of course, this requires that providers agree to expose their data through publisher services.

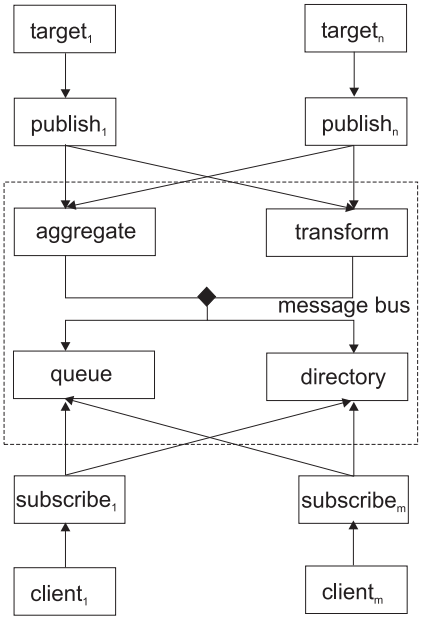

Figure 5.9: Publish-subscribe pattern

5.10 Producer-Consumer Pattern

At the end, a business use case pattern is introduced, called producer-consumer pattern. It is an adaptation of the classical producer/consumer problem. It comprises producer and consumer services. The former receives input data, processes it and puts it into the queue. Consumer takes values from the queue, performs it own processing and sends data to the output. They are both executing asynchronously, that is, they do not wait for the reply from the queue, but take the data at their own pace. Producer and consumer model two elements of a business logic. For example, producer can receive requests for bank transfer, pre-process them (authorization, feasibility) and then put a request in a persistent storage (modelled by the queue) for producer to take and perform the actual transfer and generate the report. This part of the workflow can involve human interaction, too. The entire process is executing inside a loop controlled by the external service *start*.

The producer-consumer pattern can be expressed (Figure 5.10):

$$start \circlearrowleft_{(exit>0)} (producer \triangleright queue)||(queue \triangleright consumer)$$

Let us consider the basic problems that can occur when using this pattern. Since producer and consumer are not synchronized, it is important that

producer does not put elements into the full queue, or that consumer tries to take an element from the empty queue. These things are very difficult to ensure in practice. The entire verification example based on the producer-consumer pattern can be found in Appendix D, and it shows the full strenght of both design time verification and usage of standard design patterns.

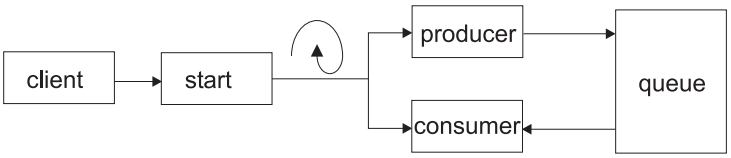

Figure 5.10: Producer-consumer Pattern

A design solution for the producer-consumer problem, with some of the already mentioned design patterns applied, can be expressed in the following form (also shown in Figure 5.11):

$$client \rhd encrypt \rhd https \rhd channel \rhd decrypt \rhd facade \rhd$$
$$(auth \Box access \Box integrity) \rhd (proxy_1 || conf) \rhd integrator$$
$$start \circlearrowleft_{(exit>0)} [(proxy_2 || log_1) \rhd producer \rhd (proxy_3 \odot proxy_4 \odot proxy_5) \rhd$$
$$(channel_1 \odot channel_2) \rhd queue] ||| [queue \rhd (proxy_6 || log_2) \rhd channel_3 \rhd consumer]$$

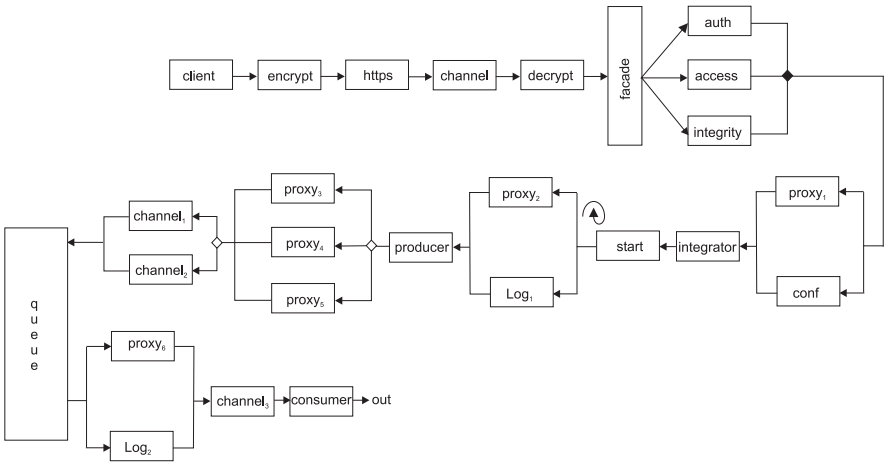

Figure 5.11: Composite Use of Design Patterns

Chapter 6

Automatic Service Composition

This chapter presents methods that have been developed for automatic service composition. It is a composition where only starting state (available services) and goal state (target service) are known. The problem is how to determine which composition matches the target service, without explicitly giving either the partner services or composition operators connecting them. This issue is relevant for dynamic and on-demand reconfiguration of service-based applications. The problem is treated as a search problem and the following search strategies are presented: basic heuristic search, probabilistic search, learning-based search, decomposition and hybrid bidirectional search.

6.1 The Need for Automatic Composition

Up to this point a framework has been described that facilitates discovery and composition of Web Services while alleviating the main disadvantage of existing methods: inability to describe non-functional properties and verify composition correctness. The proposed framework can be used to build tools that will speed up development of service-based applications and make it more secure and less error prone. The true expected value of Web Services is, however, in automated business to business (B2B) interactions, where many services belonging to different organizations dynamically cooperate in solving complex tasks [8, 19, 113].

In order to be able to properly motivate necessity for automatic service composition, a brief overview of current software and application development architectures will be made, trying to establish their shortcomings, and showing how migration to service-oriented architectures with support for automatic service composition can respond to the growing needs.

In the last decade or so, enterprise computing and application develop-

ment was (and to some extent still is) dominated by n-tiered applications, as shown in Figure 6.1. Application architecture comprises several layers: presentation (interface), business logic and data access. The particular design is also known as model-view-controller [94]. This type of architecture was created as an answer to several problems that can arise when applications contain a mixture of data access, business logic and presentation code. Such applications are difficult to maintain, because interdependencies between components create strong ripple effects whenever a change is made to any part of the system. N-tiered architecture solves these problems by decoupling data access, business logic and presentation components.

Still, even with advanced component frameworks (e.g., J2EE, .NET) and various design architectures and patterns, we still do not see a market for reusable software components. Although advanced application servers have been around for years, designers and developers are still mostly left to themselves to develop, test and deploy custom components, rewriting code and solving the same problems all over again. It is a common concern whether a component (or a service) marketplace is a myth or reality. Several reasons have been identified causing independent software vendors not to ship components to the market: maturity, politics and questionable values [147]. Since components 'live' in application servers, application servers themselves must be mature enough before a market for components written for those servers appears. Then, there is the question of proprietary application servers, and some providers (quite wrongly) see this as a competitive advantage, resulting in non-compatible application servers or intentional information withdrawal. Finally, there is no metric to determine how good a component is, or does it really fit client's requirements.

Furthermore, market globalization and the ubiquity of the Internet is forcing enterprises to abandon their heritage business models and legacy systems with business workflows deeply embedded inside business logic layer, and organize themselves into virtual enterprises [166]. On demand creation of virtual enterprises can shorten delivery times, increase product quality, deliver personalized services, decrease transaction costs, and accommodate short-term cooperating relationships, which can be as brief as a single business transaction. The two major attributes required for such environment are *extensibility* and *adaptivity*.

This paradigm requires a shift from tightly coupled business components isolated deep within the middle layer to more flexible and loosely coupled ones [174]. The migration is shown in Figure 6.1 where previously isolated business logic components now dynamically interact with each other through automatic composition of fine-grained services in ways that were not predefined and/or predicted in deployment time. It is clear that in open envi-

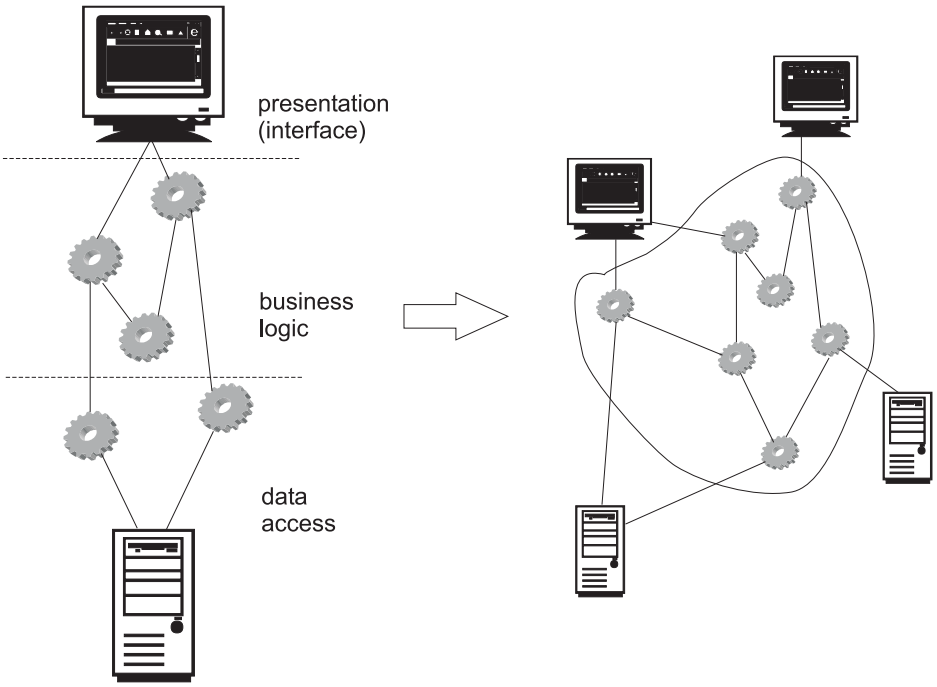

Figure 6.1: Migration from N-tier Applications to SOA

ronment like this, where services dynamically interact with each other on demand, being able to ensure correctness (dependability, security, timeliness) plays a crucial role. Web Service architecture is considered a solution that can support extensibility and adaptivity of dynamic composition [184].

Certainly, it is not (yet) realistic to expect that all business transactions will be performed by automatic composition of freely available services, for several reasons: migration from n-tiered to service-oriented architectures will be a gradual one, and even when completed, certain mission-critical or confidential business components will always remain isolated. In such cases interoperation between enterprises will be performed by merging their business workflows and applications. While maintenance and upgrades of a single n-tiered application are relatively easy, interconnecting multiple n-tiered applications is very difficult, because of their monolithic nature. Application servers are frequently incompatible even when hosting components developed using the same technologies (e.g., J2EE application servers have non-standard configuration files), and interfacing business components from the middle layer is difficult and cannot be performed in a standard manner, since component interfaces are developed by different providers in individual and application-specific manner.

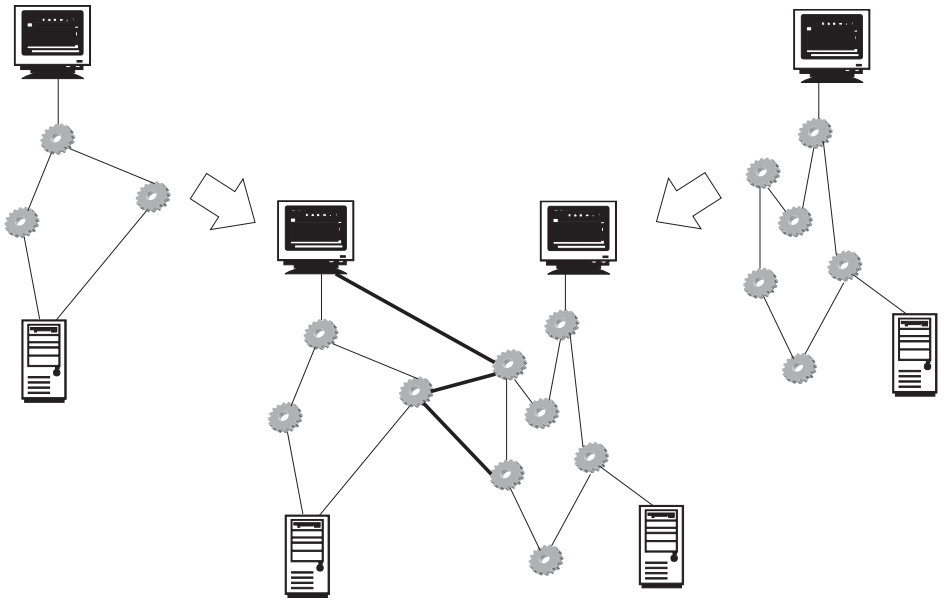

Figure 6.2: Interconnecting Two Service-oriented Applications

Integration of service-oriented applications is, on the other hand, much easier (Figure 6.2). Since all services building one application have uniform and standardized interface it is easy to interconnect required components in order to achieve desired flow. Regardless whether a service is accessed from presentation, data access or business logic layer itself, the code is the same. Figure 6.2 shows how existing links (compositions) are persisted in the integrated application, while adding connections (shown in bold) between presentation and business logic components, which describe new application logic. It is irrelevant which components are being composed, as long as description language is expressive enough that it allows for precise matching of requirements. Otherwise, the same problem that has been solved by n-tiered architectures appears again, namely ripple effects caused by modifying/composing certain parts of the system.

By automatic composition of individual services or dynamic interconnection of whole service-oriented applications it is possible to create applications with distributed data access and storage facilities, distributed business logic and distributed presentation capabilities. All these attributes are required for the new economy, characterized by brief, dynamic and flexible relationships between enterprises, infrastructures and end users. Achieving the same properties using traditional monolithic tiered approach is very difficult, if at all possible.

6.2 Equality of Abstract Machines

The problem of automatic service composition can be defined as follows: given the sets of available services E and composition operators O, and target (goal) service $t \notin E$, find the composition $e(e_1, ..., e_n, o_1, ..., o_m)$ where $e_1, ..., e_n \in E$ and $o_1, ..., o_m \in O$, such that $correct(e) = true$, and $e \equiv t$, where $t \notin E$. In other words, the task of automatic composition for a given target (goal) service t is to find adequate composition based on available services and operators that will produce a correct service e *equivalent* to t. Machine equivalence is treated as syntax equivalence only. Two machines are equivalent if and only if after renaming machine clauses (state variables and operation names) two identical machines are obtained.

However, information whether two machines are equivalent is not particularly useful on its own. In the process of automatic composition it is more important to know how two machines differ, and to be able to quantify their difference. Therefore, metrics for calculating distance between two abstract machines is introduced.

Distance between two abstract machines is a number of dimensions and substitutions they differ in. Lexical differences are not taken into account, that is, it is allowed to rename clauses of one machine. Function δ calculates the distance between machines m_1 and m_2, normalized to evaluate between 0 (equality) and 1 (complete difference):

$$\delta(m_1, m_2) = \frac{1}{2\alpha} \sum_{i=1}^{\alpha} \delta_d(d_{1i}, d_{2i}) + \frac{1}{2\beta} \sum_{j=1}^{\beta} \delta_s(s_{1j}, s_{2j})$$

where $|m|_d$ is the number of machine dimensions (state variables, machine formal parameters and constants), $|m|_s$ is the number of substitutions that make operation body (pre-conditions, post-conditions, invariants), $\alpha = max(|m|_{1d}, |m|_{2d})$, $\beta = max(|m|_{1s}, |m|_{2s})$ and δ_d and δ_s calculate number of differing dimensions and substitutions:

$$\delta_d(d_1, d_2) = \begin{cases} 0 & d_1 = d_2 \\ 1 & d_1 \neq d_2 \end{cases}$$

$$\delta_s(s_1, s_2) = \begin{cases} 0 & s_1 = s_2 \\ 1 & s_1 \neq s_2 \end{cases}$$

Two dimensions are equivalent if and only if their types and directions (state variables) are equal. Dimensions are compared using their types and directions, not their names. In that respect, variables `input` and `in` are equivalent:

```
MACHINE A
SETS InputSet={x,y}
VARIABLES input:IN
OPERATION doSomething PRE input ∈ InputSet
...
END

MACHINE B
SETS InputSet={x,y}
VARIABLES in:IN
OPERATION doSomething PRE in ∈ InputSet
...
END
```

Although they have different names, these variables have the same direction (IN) and their type-checking evaluates to the same set: type(input)= InputSet and type(in)=InputSet. Constants and machine formal parameters also constitute dimensions and their properties and constraints are implicitly taken into account in the process of type checking the same way that invariants and pre-conditions are used to type state variables.

Two substitutions are equivalent if and only if they perform the same substitution on equivalent state variables in the equivalent order. Consequently, two operations are equivalent if and only if all their substitutions are equivalent. The following two machines have equivalent operations:

```
MACHINE A
SETS InputSet, OutputSet
VARIABLES input:IN, output:OUT
INVARIANT output ∈ OutputSet
OPERATION output <- doSomething(input)
PRE input ∈ InputSet THEN
input := input + 1 □  input := input - 1;
output := input END
END

MACHINE B
SETS InputSet, OutputSet
VARIABLES int:IN, out:OUT
INVARIANT out ∈ OutputSet
OPERATION out <- doSomething(in)
PRE in ∈ InputSet THEN
in := in - 1 □  in := in + 1;
out := in END
END
```

For multiple generalized substitutions and choice substitution, the operand order is irrelevant. That is the reason why operations of machines `A` and `B` are equivalent, even if the order of substitutions under choice clearly differs. On the other hand, if operation of machine `B` is defined as:

```
OPERATION out <- doSomething(in)
PRE in ∈ InputSet THEN
out := in;
in := in - 1 □  in := in + 1 END
END
```

Operations are not equivalent anymore since they differ in the order of two substitutions.

Generally speaking, judging difference of substitutions is far more difficult when compared to difference of dimensions. It is possible to develop a more precise measure of substitution difference by taking into account the actual substitution type and creating a function that is not purely binary, but offers a finer measurement of substitution equality. Therefore, function δ_s is modified as follows:

$$\delta_s(s_1, s_2) = \frac{1}{2} weight(s_1, s_2) + \frac{1}{2\gamma} \sum_{k=1}^{\gamma} \delta_d(d_{1k}, d_{2k})$$

The second part of this function simply calculates the number of dimensions that two substitutions s_1 and s_2 differ in, where $\gamma = max(|s|_{1d}, |s|_{2d})$ and $|s|_d$ is the number of dimensions of substitution s. The first part is the weighted function that describes a semantic difference between substitution types. The function *weight* is given in Figure 6.3.

Value κ is used to compare two exact substitutions that may differ in their predicates. For example, it makes sense to give two exact preconditions score 0 (equality), and score 0.5 otherwise, since two preconditions differ less than pre-condition and while substitution for example. The remaining difference in predicates will be eventually calculated by the second part of the function (δ_d). Therefore, κ is defined:

$$\kappa = \begin{cases} 0 & \text{predicates equal} \\ 0.5 & \text{otherwise} \end{cases}$$

Similarly, when comparing two multiple or choice substitutions, the value of 1 could be assigned if they differ, but again additional measure is introduced to soften this criteria by comparing number of operators (|| or □) in which they differ. Therefore, μ is introduced:

$$\mu = 1 - \frac{min(|s_1|_{op}, |s_2|_{op})}{max(|s_1|_{op}, |s_2|_{op})}$$

	$S;T$	$S\|\|T, S \odot T$	PRE	CHOICE	IF	ELSE	ANY	WHILE
$S;T$	0							
$S\|\|T, S \odot T$	0.6	μ						
PRE	1	1	κ					
CHOICE	1	1	1	μ				
IF	1	1	0.5	1	κ			
ELSE	1	1	0.75	1	0.5	κ		
ANY	1	1	0.95	0.9	0.9	0.95	κ	
WHILE	0.9	0.95	0.9	1	0.9	0.95	0.95	κ

Figure 6.3: Weight of Substitutions

Values in the weight table are not fixed, nor do they have any constraints imposed upon them (except the obvious one that all must be less or equal to one). Each value is inductively developed by observing behavior of different substitutions. For example, it is obvious that the following two substitutions (sequential and parallel) are not equal in any sense:

```
x := x + 1;
y := x

x := x + 1 || y := x
```

On the other hand, situation is different if variables are independent:

```
x := x + 1;
y := y + 3

x := x + 1 || y := y + 3
```

Functionally speaking, these two substitutions are equivalent. At the end of their execution, values of `x` and `y` will be the same. Of course, the are not completely semantically equivalent, as their execution time, for example, may not be equal. Therefore, a value has to be picked that represents a probability that sequential and parallel substitution can be considered equal and therefore treated more favorable than for example `IF` and parallel substitutions which are clearly always different. The value that has been selected

is 0.6. Again, this is a subjective value, which was developed in an inductive process of examining all possible substitution reductions.

Another example is comparing `WHILE` and parallel substitution. While can be reduced to parallel, but in a very extreme and improbable case:

```
WHILE x < 1
DO
 x := 2;
 y := 3;
 z := 1
END

x := 2 || y := 3 || z := 1
```

This kind of reduction is very unlikely, therefore value of 0.95 is assigned to this case. Similar reasoning is used to obtain all other values in the weight table.

6.3 Modeling State Space

The problem of automatic service composition is essentially a search problem [112]. To be able to formulate strategies for automatic composition of abstract machines, certain elements need to be defined for designing adequate search methods:

- *State space* containing all possible configurations of the objects upon which a search is performed. State space comprises atomic services and all correct composition of thereof.
- *Starting state*, which is one or more states from state space that describe possible situations (configurations) from which a search can start. These states are also called initial states. Starting/initial states are atomic services.
- *Goal state* is one or more states that can be accepted as a solution of a search. End state is a target service.
- Finally, a set of *rules* that describe the actions or operations that are available for transforming initial state towards goal state. In our case rules are obviously composition patterns.

Some properties of this problem will be examined, based on which search strategies will be decided [141]. The problem of automatic service composition is decomposable, under the assumption that target service (machine) is

correct. A problem is decomposable if it can be transformed into a set of independent smaller or easier subproblems. Typical example of decomposable problem is symbolic integration. In that sense service composition can be treated as decomposable, but the practical applicability of decomposability is somewhat limited. For example, request for a service that makes flight and hotel room reservation can be decomposed into two subproblems: hotel reservation and flight reservation. Such decomposition, however, is neither always obvious and/or easy to identify nor services required to perform it are available. Decomposition, as a divide-and-conquer methodology, will be investigated in more detail in Section 6.7.

The problem universe is predictable. Predictable problem universe is the one in which application of rules has a certain outcome. Indeed, predictability of composition properties was one of the main reasons for introducing abstract machines and composition patterns. It is always known what will be the exact result of applying a certain rule (operator) to the current state (composition). In other words, every time a move is made in the state space, the following, resulting state is precisely known. This means that an entire sequence of moves can be planned in advance. However, this is true only if trust is assumed. As already discussed, service contracts are composed trusting them to be correct and accurate representation of relevant properties of underlying implementation.

The rules application is recoverable. This means that we can go back if a certain search path is misleading, but we will need to backtrack to a certain point since rule application cannot just be ignored: a part of a solution will have to be "undone" or "uncomposed". For example, if one hotel reservation service is composed with one flight reservation service and it is then found out that flight reservation service does not support transactions causing the entire solution not being able to execute in one transaction, it may be decided to drop the particular flight reservation service and try to locate another one. However, once another candidate has been located, it cannnot just be composed on top of previous solution. We must first backtrack to the point in state space where previous flight reservation has been composed. This means that adequate control structure must be introduced to enable backtracking. The simplest way to do this is to use a stack to record rule (composition patterns) applications.

Goal solution is absolute, assuming equality of machines is defined. Once a satisfactory solution has been found, the search can be stopped. That means that it is not needed to search further and compare multiple solutions, since only an equivalent solution can be found. This is only true, however, if an absolute solution can be found. Otherwise, suboptimal solution can be negotiated. For example, if a travel reservation system cannot locate both

hotel and flight for a given price, it can offer the next best (although more expensive) solution to the user.

Rules are consistent, assuming that composition operators are well defined and proved. In this case that means that it is allowed to use only the operators that were previously defined (sequence, selection, choice, parallel and loop composition) to move through the state space. Under this assumption, problem is consistent. This is only true if no additional knowledge is being used for reaching the goal. Otherwise, as the following sections will show, special attention must be paid that additional knowledge is also consistent. The simplest example of inconsistent knowledge is when knowledge base contains both A and $\neg A$.

We aim for a solution where no intermediate interaction with the end (human) user will be required. In case that two machine entities are communicating and trying to compose new service, such interaction is also not necessary. Human interaction can be required for two reasons: to provide additional input during the search process or to provide additional reassurance and justification of the solution to the user. If a solution cannot be found, a solitary way of solving automatic composition problem can be transformed into a conversational one, where end user is offered sub-optimal solution on one or more criteria, and has to accept this solution explicitly via some sort of interface. This should not, however, be the basic mode of operation.

Finally, since the objective of automatic composition is to find a path through a state space connecting starting state and goal state, there are two directions in which the search can proceed: moving from starting state towards goal state (forward search) or moving from goal state towards starting state (backward search). When deciding which strategy to use, number of start and goal states is usually taken into account. It is preferable to move from the smaller set of states to the larger (thus easier to find) set of states. It is also good practice to move in the direction of the lower branching factor. The branching factor is the average number of states that can be reached directly from a single state. In both respects backward search seems to be more appropriate, since branching factor can be only equal or less when compared with forward search and number of goal states is certainly smaller. However, applying composition patterns in the *reverse* order to decompose an abstract machine is not trivial. Therefore three forward search mechanisms will be first investigated and developed, then a way to decompose abstract state machines will be introduced and finally a hybrid bidirectional solution will be proposed. Based on these observations, the following search strategies are formulated:

- Basic heuristic search of state space (forward search)

- Probabilistic automatic composition
- Automatic composition by learning
- Decomposition (backward search)
- Hybrid bidirectional search

Before proceeding to search methodologies, state space elements and traversals must be defined. Basically, there are two choices: to model state space as a tree or as a graph (with option to switch between the two).

One form of a tree that can be used to represent state space transitions is a syntax tree [5]. In a syntax tree, inner nodes represent composition patterns and leaf nodes represent services (abstract machines). Examples of some syntax trees for services A, B and C and composition patterns $\triangleright$ and $||$ are given in Figure 6.4.

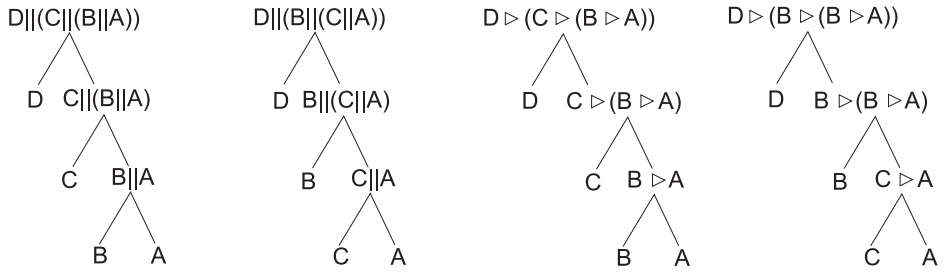

Figure 6.4: Syntax Tree

The benefit of using syntax tree to model state space is that a syntax tree can be converted to deterministic/nondeterministic finite automaton. Such an automaton can be queried whether a given composition can be found in a syntax tree. The problem is, however, that this would require that entire (or at least a significant part of) state space is already expanded in a syntax tree. Syntax trees are not well balanced and addition of new nodes and elimination of duplicate ones are not trivial or cheap operations. An alternative to tree is to represent a state space as a graph.

The graph form that will be used is similar to AND-OR graphs which consist of OR edges and AND arcs, where one AND arc can point to any number of successor nodes. It is used primarily to represent problems that can be decomposed into smaller problems connected by AND arcs that must all be solved in order for the original problem to be satisfied. Example of an AND-OR graph is shown in Figure 6.5. The possible paths from node *acquire new car* are: *steal a car* OR *earn enough money* AND then *buy a*

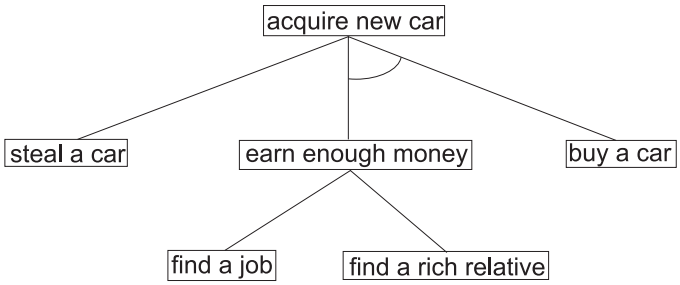

Figure 6.5: And/Or Graph

car. From node *earn enough money* it is possible to go to either *find a job* OR *find a rich relative*. Similar idea is used to allow multiple arced edges to connect nodes that are composed using given composition pattern. Instead of AND operation, an arc represents a composition pattern (Figure 6.6).

The benefit of using graph representation is that new nodes are added easily, as can bee seen with composition resulting in $(A||B||C||D) \rhd (D \rhd C \rhd B \rhd A)$. It is also possible to define operand order using right-hand rule. Naturally, right-hand rule serves only to eliminate possible ambiguities in graphic representation and does not carry any other deeper meaning, since this is not a geometric graph.

Naturally, there is an option to switch to a search forest. For every atomic service a tree is created with the atomic service as its root by taking all graph nodes in which that atomic service is the left-most operand, removing all nodes representing right-most operands, and containing operators and right-most operands implicitly in the graph arcs, thus producing a search forest shown in Figure 6.7. Both representations will be used interchangeably.

The problem that needs to be solved is how to construct and move through such composition graph/forest in order to find desired composition. The next sections present several search strategies.

6.4 Basic Heuristic Automatic Composition

It should be obvious that brute force search, where all possible combinations of services and composition operators are explored until a composition matching the target is found, is unrealistic because combinatorial explosion renders it impractical. The problem is the number of available services, as well as the number of composition operators allowed. The fact that operators can be n-ary and that same service can appear in a composition more than once further complicates any kind of non-heuristic search.

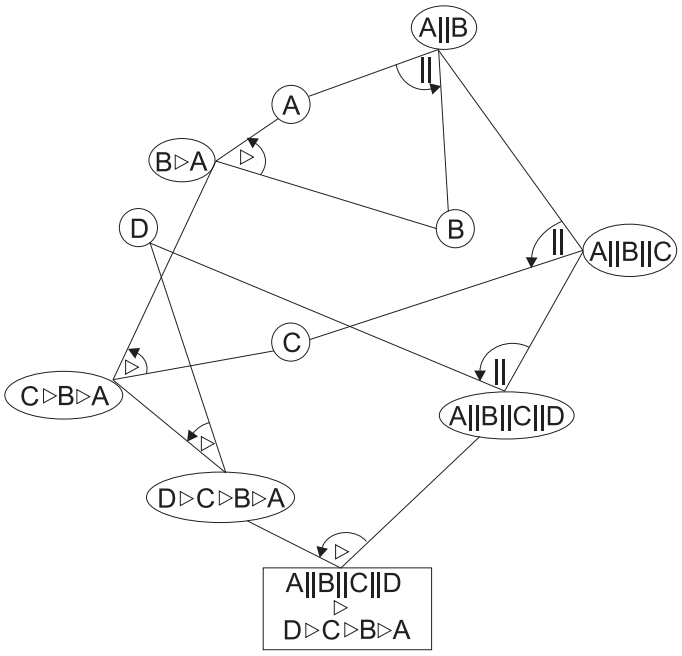

Figure 6.6: Composition Graph

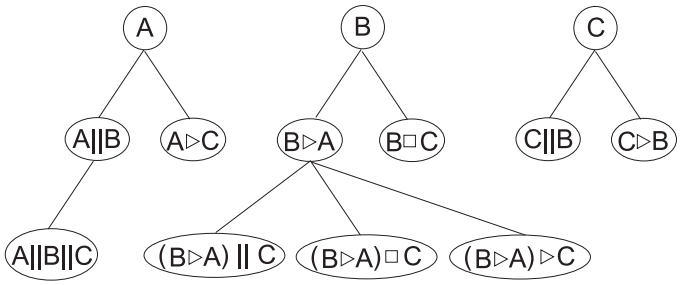

Figure 6.7: Part of a Search Forest

Two well-known "weak" methods will be first discussed, that require no additional heuristics: depth-first search and breadth-first search [178] which systematically generate correct compositions starting from available services and operators. Both algorithms deteriorate rapidly with the expansion of state space, former with increasing number of services and latter with increasing number of composition operators. Therefore, some additional knowledge of the problem domain is necessary. A simple heuristic for abandoning a certain branch is then introduced:

- If more states are generated that the target machine has, abandon

the branch since further application of composition operators can only increase or leave the number of states unchanged

- If composition operator is commutative and equivalent branch has been explored, abandon the branch.
- If composition operator is associative, and one association has been explored, ignore the branch with other association.
- If composition operator is distributive, and either distributed or condensed formula has already been applied, ignore the branch with the other one.

This heuristics improves the performance, but has problems if many composition operators are introduced, especially if they are *not* commutative, associative or distributive. Therefore we try to develop appropriate heuristic function. A heuristic function maps from problem state description to measure of desirability (usually quantified as number). That means that for each element of state space it gives quantitative measure how close that state is to a solution. The purpose of a heuristic function is to guide the search process in the most promising (profitable) direction. It does so by suggesting which path to follow through the state space when more than one is available. The more accurate heuristic function is (the more accurately it evaluates the merit of each state), the faster and more direct will be the whole search process. Two well known heuristic approaches are A* [65, 66] and AO* [95, 96, 125] search algorithms. They will be used as basis for developing a range of heuristic search approaches for automatic composition of abstract machines. The reason why these two methods have been selected as starting points is in the way the state space and knowledge is modeled. Neither A* nor AO* can perform a search for the graph that is used to model service composition, since A* can be used for OR graph and AO* for AND-OR graph traversal only. However, problem properties (as described in the previous section), as well as the similarity between the AND-OR graph and the composition graph justify the assumption of problem compatibility.

Before proceeding to the description of the search algorithm, several elements will be introduced. Since state space is infinite, a measure of *futility* needs to be introduced in order to cut search paths that will never lead to the result. Therefore a value F will be a measure for the futility of a given search path. Different measures for futility can be accepted: it can be a value of heuristic function that shows that a distance to the goal is too big to be realistically reached, or a number of current solution's dimension which can be too large to fit into the goal machine. In any case, F must be such that it

can guarantee that abandoning any search path will not result in a solution being missed, that is, that subsequent composition will not change the value of F so it becomes favorable again.

During the execution of the algorithm, three lists are maintained:

- `OPEN` contains nodes that have been generated and heuristic function has been applied to them, but they have not yet been expanded, that is, their successors have not yet been generated.
- `CLOSED` contains nodes that have already been examined and expanded, and have not crossed futility value.
- `LIMIT` contains expanded nodes that have crossed futility value.

Finally for every `CURRENT` node that is being expanded a heuristic function f' is given with:

$$f'(\texttt{CURRENT}) = \delta(\texttt{CURRENT}, \texttt{GOAL})$$

where `GOAL` is the node representing composition target (search goal), and δ is the distance function given in Section 6.2. It will be used to select the nodes that are closest to the goal node (with the smallest value of δ) to be expanded first.

Apart from using futility F and heuristic function δ to guide heuristic search, it is also important to detect and cut off equivalent search paths as early as possible. For example, it is obvious that compositions $A||B$ and $B||A$ are equivalent, yet they can appear more than once in a search forest. All subsequent compositions based on these two nodes would be also equivalent, therefore one of the nodes can be safely removed. In order to deal with this issue, several definitions and rules are introduced that enable detection and handling of such cases:

1. Every abstract machine is a term.
2. If A and B are terms, than $A \triangleright B$, $A||B$, $A \Box B$, $A||_P B$, $A \odot_C B$, $A \circlearrowleft_P B$ are also terms.
3. Every abstract machine is equivalent to itself.
4. Two terms $A_1 \circ B_1$ and $B_2 \circ A_2$ are equivalent if and only if:
   - operator $\circ$ is commutative and,
   - A_1 is equivalent to A_2 and B_1 is equivalent to B_2, or
   - A_1 is equivalent to B_2 and B_1 is equivalent to A_2.

5. Two terms $A_1 \circ (B_1 * C_1)$ and $(A_2 \circ B_2) * C_2$ are equivalent if and only if:

   - operators $\circ$ and $*$ are associative, and
   - A_1 is equivalent to A_2, B_1 is equivalent to B_2 and C_1 is equivalent to C_2.

6. Two terms $A_1 \circ (B_1 * C_1)$ and $(A_2 \circ B_2) * (A_2 \circ C_2)$ are equivalent if and only if:

   - operator $\circ$ is distributive with respect to $*$, and
   - A_1 is equivalent to A_2, B_1 is equivalent to B_2 and C_1 is equivalent to C_2.

Finally, for every node function g is defined that is used for algorithm termination, if the value of g exceeds the threshold value F. Function g is accumulated during the forest traversal, and for every `NEW` node is given by:

$$g(\texttt{NEW}) = g(\texttt{CURRENT}) + \delta(\texttt{NEW}, \texttt{GOAL})$$

Next, the functioning of the algorithm will be described. At the start of the algorithm, list `OPEN` contains all atomic services, and all other lists are empty. F is assigned an initial value. The search will go on until a goal abstract machine is reached, or all subsequent expansions cross the limit F. For each node in `OPEN`, function δ is calculated and stored. The node with the smallest value of δ, that is, the node that is closest to the goal machine is chosen for composition. If more nodes have the same value of δ, choice is performed randomly. The chosen node is then composed with all nodes from `OPEN` and `CLOSED` using all operators. This is the expansion phase, where node successors are generated. The current node that is being expanded is always the leftmost operand. If any of the generated nodes is equivalent to the goal machine, search is ended. For each new node function g is calculated and compared to futility value F. In case $g > F$ the node is too far away from the goal to be considered further and the cut off is performed by storing the node in `LIMIT`. Each new node is also compared to all generated nodes that are stored in `OPEN`, `CLOSED` or `LIMIT` using equality rules. This step ensures that equivalent paths are detected and cut off. Finally, if `OPEN` is empty and no solution has been found yet, nodes that have not been considered (`CLOSED`) are moved to `OPEN` and expanded. A complete and detailed step-by-step example of the algorithm execution can be found in [86], while formal algorithm description is:

1. OPEN contains the atomic services only. CLOSED and LIMIT are empty. Value of function g for every atomic service is 0. F is given an initial value.

2. Until the goal is reached or OPEN and CLOSED are empty, the following steps are repeated:

   - The node with the smallest value of δ is chosen from OPEN, assigned identifier CURRENT and removed from OPEN.
   - If CURRENT is equivalent to GOAL, CURRENT is returned and search is ended.
   - Otherwise, successors of CURRENT are generated. For each NODE in OPEN, CLOSED and {CURRENT} the following is performed:
     (a) Node NEW is generated from CURRENT, operator and NODE.
     (b) If NEW is a solution, return NEW and exit.
     (c) $g(\texttt{NEW}) = g(\texttt{CURRENT}) + \delta(\texttt{NEW}, \texttt{GOAL})$ is calculated.
     (d) If there is no equivalent node to NEW in OPEN, CLOSED or LIMIT, $g(\texttt{NEW})$ is compared to F. If $g(\texttt{NEW}) > F$, NEW is put to LIMIT, otherwise to OPEN.
   - If no new nodes are added to OPEN, that is, all successors of CURRENT have g value larger than F, CURRENT is put to LIMIT, otherwise to CLOSED.
   - If OPEN is empty, exchange OPEN and CLOSED.

The best way to guarantee efficient and fast heuristic algorithm is to have quality heuristic function that will guide the search in the most promising direction. Various additions to the heuristic function presented here will try to improve this result by using more efficient heuristic functions. They will always have more favorable average execution complexity than this approach, as they will fall back to the basic heuristic search in case the solution has not been found using advanced heuristics.

6.5 Probabilistic Automatic Composition

Heuristic function from the previous section uses distance between abstract machines as a measurement which branch is most promising to follow. In this section that heuristic is augmented with the idea of probabilistic search.

The additional heuristic is represented as weighted directed graph with vertices representing services and edges representing composition patterns,

as shown in Figure 6.8. Each edge is assigned a probability that a service from which an edge is originating will cooperate with a service in which the edge is ending. The sum of all outgoing weights for any node must be less or equal to one. If equal to one, all possible interactions for a given service are known (which is very unlikely). Otherwise, there is a possibility that unknown services can cooperate with a given one, with probability of $1 - \sum_{i=1}^{k} w_i$, where w_i is weight of an outgoing edge and k is number of outgoing edges.

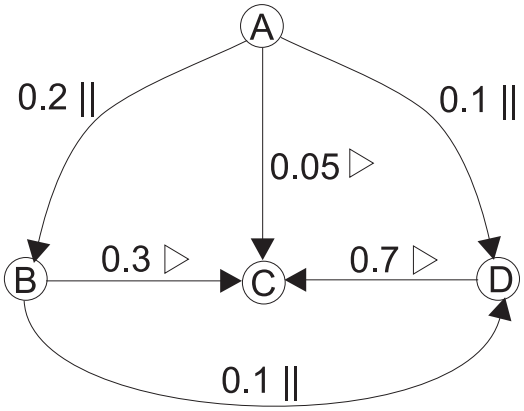

Figure 6.8: Cooperation Graph

The probability of a branch of length k is calculated by multiplying probabilities of constituent edges:

$$P(B_k) = \prod_{i=1}^{k} w_i$$

For example, if we are looking for sequential compositions ending with service C from Figure 6.8, there are four possible branches: $A \triangleright C$, $A||B \triangleright C$, $A||B||D \triangleright C$ and $A||D \triangleright C$, where $||$ and $\triangleright$ denote parallel and sequential composition respectively. Probabilities of identified branches are 0.05, 0.06, 0.014 and 0.07, therefore a path $A||D \triangleright C$ is chosen.

This assumes, however, that events of choosing next cooperating service are independent. In our example there is no difference whether we arrived at node D from A or B: node C will be subsequently picked with 0.7 probability. Therefore causality is introduced by adding conditional probabilities. Assume A and B are two events, then:

$$P(AB) = P(A|B)P(B)$$

where $P(A|B)$ denotes probability of event A under the assumption that event B took place, while $P(AB)$ is the probability that both events occurred. Furthermore, if $A_1, ..., A_n$ are events, the following holds:

$$P(A_1A_2...A_n) = P(A_1)P(A_2|A_1)P(A_3|A_1A_2)...P(A_n|A_1A_2...A_{n-1})$$

Using these results additional conditional probabilities are created in the cooperation graph that describe causality effect of choosing previous nodes (Figure 6.9). Let us assume that $P(D \rhd C|A) = 0.1$ and $P(D \rhd C|B) = 0.6$, that is, we now distinguish between cases $D \rhd C$ when we arrive to D from A and from B. Now $P(A||D \rhd C) = P(A||D)P(D \rhd C|A) = 0.01$ and $P(A||B||D \rhd C) = P(A||B)P(B||D)P(D \rhd C|B) = 0.012$. The most favorable path now changes to $A||B \rhd C$. Using formula for n-conditional events we could go deeper into the cooperation graph. However, adding even this one level of causality provides a significant improvement compared to graph with independent probabilities.

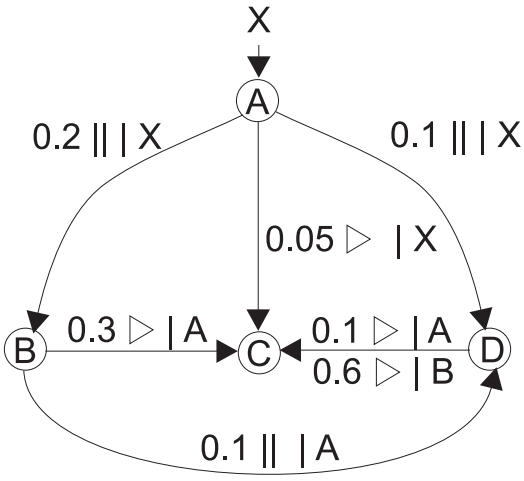

Figure 6.9: Causal Cooperation Graph

By creating cooperation (probability) graphs, implicit human knowledge of a state space properties is exploited, by assigning higher probabilities to combinations that are more likely to work out together. For example, a stock ticker service is more likely to cooperate with a stock trading or a printing service than with a book searching service, although all combinations are functionally possible. Fixed probabilities however are not very realistic. Therefore the approach is made more flexible by allowing probabilities to

change over time. In an adaptive process, probabilities of branches (compositions) that are used more frequently are increased and vice versa. This change is made for all edges in a branch, while assuring that sum of all edges originating from any node in a branch does not exceed 1. For all composition patterns two tables are maintained: *compositions* and *probabilities*. In *compositions* rows and columns represent services and entries number of successful compositions. Table *probabilities* has the same structure and at the beginning is populated with initial probabilities. After assigning initial probabilities $P_{init}(N_i, N_j, op)$, *compositions* table entry (N_i, N_j, op) is incremented when N_i and N_j are composed using pattern op. Total number of compositions for each pair (n) is also maintained. After each composition, current probability is calculated $P_{current}(N_i, N_j, op) = k(N_i, N_j, op)/n$, where k is *compositions* table entry for (N_i, N_j, op). This probability is not automatically stored in the table *probabilities*. If any value in row N_i becomes such that $|P_{init}(N_i, N_j, op) - P_{current}(N_i, N_j, op)| > \epsilon$, probabilities of entire row are recalculated and stored in *probabilities*: $P_{new}(N_i, N_j, op) = P_{current}(N_i, N_j, op)$. To prevent fast (and non-realistic) oscillations, comparison of $P_{current}$ and values from *probabilities* can be done periodically and not every time *compositions* is updated. Also, table *compositions* can be reset, with current probabilities accepted as new initial probabilities. Value of ϵ can also change if necessary.

The quality of this approach depends on the way initial probabilities are assigned. Initial probabilities are important for two reasons: they determine starting conditions under which services compete, and can also be used when reseting *compositions* table, if for some reason one does not want to use current probabilities. If initial probabilities are not realistic, convergence to optimal balance can take a lot of time and render the whole approach unusable. Therefore a method for assigning initial probabilities is proposed, using service classification.

Figure 6.10 shows layered service classification comprising physical, network and application service classes. Classes can communicate to neighboring classes only, e.g., service of class physical can communicate to network class only, network can communicate to both physical and application, while application can interact with network class. This does not mean that actual interaction among physical class and application class is not possible, but only that in the process of distributing initial probabilities such interactions are not taken into account. There is a special subclass of all three classes called agent. Agents represent generic properties of each class and can interact with each other across any class. For example, memory agent of the physical class interacts with convert agent of the application class with probability 0.3 (Figure 6.10).

Initial probabilities can be assigned directly in the classification, or as a set of rules. Rules have the syntax (*source_ class,destination_ class,probability*). Rule is applicable to all subclasses of a given class. A subclass can redefine one or more rules, and the new rule is then further applicable to itself and its subclasses. Cooperation graphs define fine-grained interaction between services as they determine probabilities that particular service *instances* will cooperate using particular composition patterns. Classification defines coarse-grained interaction between service *classes* (not service instances) that can be applied to all services belonging to a given class.

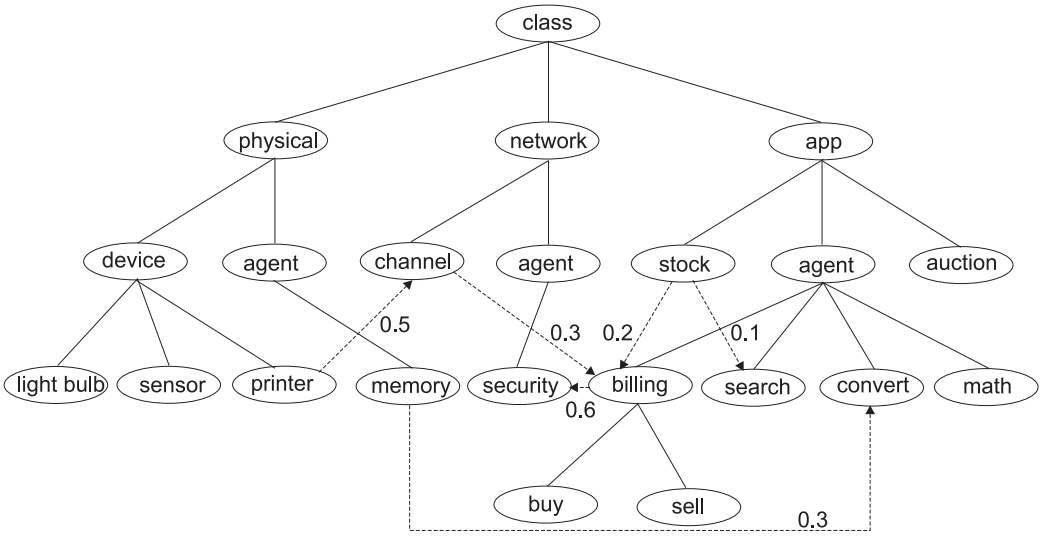

Figure 6.10: Service Classification and Initial Probabilities

6.6 Automatic Composition by Learning

The method of adaptive conditional probabilities works good when starting probabilities have favorable approximation and the weighted graph quickly converges towards optimal. When neither of these conditions are met, e.g., when nature of the requests changes frequently over time, heuristic approach from Section 6.4 achieves better results. Another possible approach for relatively stable environments (one where requests can be at least classified or typed) is learning-based composition.

The basic idea of the learning based approach is to provide the system with solutions (compositions) to the common recurring problems (goals), and then expect that the system can adapt to solving the common problems when

some of the conditions in the environment change, without the need for manual intervention (recomposition). More precise, the system is demonstrated all compositions that solve a certain class of problems, and is then asked to try to solve a problem that has distance one from the problems it knows how to solve, where distance is defined by the function δ. The key premise is that by combining 1-distance solutions, 2-distance targets can be reached quickly. Obviously this cannot be proved, nor is always possible, therefore a process of additional substitution is introduced. It is based on service hierarchy introduced in the previous section, that is developed from classification information provided in the service contract. Classification is hierarchical, and determines which services are taken into consideration for substitution. If 1-distance solutions themselves cannot provide solution, substitution will try to locate services of similar capabilities and replace some of them. An element can be substituted with either an element of the same class or an element from any of its subclass. For example, member of network agent class can be replaced by another member of the network agent class or by a member of the security class.

A simple scenario of this idea follows: suppose a printer is available that can print only postscript documents. A system is taught to solve all printing requests by sending them to the printer. This works only if the document being printed is in the appropriate format. Therefore, a system is taught how to convert other formats: there is a class of converter services that supply different types of conversions. Suppose further that a system is taught how to convert jpg image file to postscript by invoking appropriate converter service. If a system now receives a request to print a file in pdf format, it will look into all 1-distance compositions offering printing and find how to print jpg files. Then it will try to substitute a converter service with another service from the same class, or to substitute a printer service. Either way it will end up with a printer that can print pdf, or a converter service that can turn pdf into postscript.

Let us formalize the presented approach. Let T be a set of target abstract machines and S a set of all possible solutions (compositions). A system is presented a target abstract machine $t \in T$ and then demonstrated a solution $s \in S$ (possible composition), which is afterwards persisted in a directory. A system is then presented with a set of all possible target machines $\{t_1, ..., t_n\} \subseteq T$ such that $\forall m \in \{t_1, ..., t_n\} | \delta(t, m) = 1$ and demonstrated a set of solutions $\{s_1, ..., s_n\} \subseteq S$. This completes 1-distance training. If a system is then presented with a 2-distance target machine d ($\delta(t, d) = 2$), the solution is obtained as follows:

1. Create all combinations of 1-distance solutions $\{s_1, ..., s_n\} \times \{s_1, ..., s_n\}$

and see if they match d.

2. If any matches d exit, else start substitution.

   (a) For each element of newly generated solution set, consult service hierarchy and substitute one service at a time in each composition. A service may be substituted with a member of the same class or any of its subclasses.

   (b) Revalidate solution.

   (c) If new solution matches d exit, else continue with substitution until all elements in all solutions have been substituted.

3. No solution has been found, proceed to basic heuristic search, not considering branches already visited.

This approach gives best results in more controlled environments, e.g., inside an enterprise, since precise classification and ability to determine whether two services can be substituted is required.

6.7 Decomposition of Abstract Machines

Decomposition of abstract machines is backwards search methodology. It begins from the goal state represented by a target abstract machine and iteratively tries to decompose it into simpler machines connected with composition patterns until a starting state consisting of atomic (available) machines is reached. The inverse path taken from goal state to starting states is the required composition.

Decomposition can be the preferred way of doing automatic composition, since we are moving from the known goal state (match of user's requirements) to the larger (compared to goal state) and thus easier to find set of starting states (atomic abstract machines). The approach to decomposition is based on transforming target machine substitutions to the postfix-like form. The decomposition algorithm has three main phases:

1. Convert operation body to postfix representation.

   (a) Generate relevant clauses for all variables copied to the output thus creating corresponding (variables, clauses) pairs.

2. Scan finished postfix string and determine possible compositions.

   (a) Verify correctness of every variable (machine) copied to the output.

   (b) If a copied machine exists in a directory, mark it as finished and put in finished list.

3. Check composition to determine whether all elements are in the finished list.

The first phase is now described in more details. Postfix conversion is performed on target machine operation body. During conversion, operator priorities are evaluated using Table 4.6. Result of conversion are variables and operators. Variables are machine state variables; operators are either composition patterns or generalized substitutions. In the process of postfix conversion abstract machine operation body is gradually decomposed by extracting its subelements (state variables) and composition patterns that build the goal abstract machine. The process of postfix conversion consists of the following steps (function *convert*(operation_body)):

1. Let the final `END` of the target abstract machine be a terminating symbol.

2. Terminating symbol is pushed onto the stack.

3. Variables are always copied to the output.

4. Left parenthesis is always pushed onto the stack.

5. When a right parenthesis is encountered, the symbol at the top of the stack is popped off the stack and copied to the output. This is repeated until top of the stack is left parenthesis. Then both parenthesis are discarded.

6. If an operator has a higher priority that the operator at the top of the stack, it is pushed onto the stack and stack pointer is incremented.

7. If the priority of the operator is lower or equal to the operator on top of the stack, one element of the stack is popped to output. The stack pointer is not decremented. Instead the current operator is compared with the new top of the stack.

8. When the end symbol is reached, the stack is popped to the output until terminating symbol is also reached. Then the conversion terminates.

Since we operate on operation body only, we need to generate other abstract machine clauses when a variable is copied to the output. For each state variable, target machine clauses are scanned. If a state variable appears in a

given target machine clause, that clause is copied to the output and joined to the corresponding variable. Therefore, if the current variable being scanned is `ticketPrice` and there exists a pre-condition `PRE ticketPrice > 0` in the target machine, then this pre-condition is associated with the variable at the output.

Finally, based on a postfix string and the content of the finished list, adequate composition is created. This step is best explained using an example. Suppose that we want to build a composite service that takes a loan application from the client, determines its credit rating, applies for a loan with two banks and then chooses the better loan offer (the one with higher average performance index). We start by specifying target abstract machine (service):

```
MACHINE loanExample
VARIABLES application:IN, rating, offer1, offer2, result:OUT
process_application, offer_loan, wcet
SETS App, Rating, Off, Time
INVARIANT process_application ∈ App → Rating ∧ application ∈ App ∧
offer_loan ∈ Rating → Off ∧ rating ∈ Rating ∧ result ∈ Off ∧
offer1 ∈ Off ∧ offer2 ∈ Off ∧ wcet ∈ Off → Time
OPERATION result <- ask_loan(application)
PRE rating > 0 ∧ offer1 > 0 ∧ offer2 > 0 ∧
wcet(offer1) < 86400 ∧ wcet(offer2) < 86400
THEN
rating := process_application(application);
[ ( offer1 := offer_loan(rating) || offer2:=offer_loan(rating) );
 ( (offer1 > offer2) ⟹ result := offer2 □
 ¬ (offer1 > offer2) ⟹ result := offer1 ) ]
END
```

Machine accepts variable `application` representing loan application. Application is processed and as a result variable `rating` is produced representing applicant credit rating. Two loan offers, `offer1` and `offer2` are then generated in parallel by sending credit rating to two banks. After both loan offers are ready (service will wait up to 24 hours which is specified in the pre-condition), they are compared and the better one is chosen. The process of postfix conversion produces the following string (line breaks are added for clarification only):

```
rating process_appplication(application) := offer1 offer_loan(rating)
:= offer2 offer_loan(rating) := || offer1 offer2 > result offer2 := ⟹
offer1 offer2 > ¬ result offer1 := ⟹ □ ; ;
```

The postfix string is then iteratively scanned from left to right, in an

attempt to extract possible constituent abstract machines and composition patterns connecting them. Variables are scanned until a first operator is reached, which is then applied to the variables. All variables scanned in a single pass are assigned to a single abstract machine. This machine is being verified and checked for in a directory. If the obtained abstract machine exists in a directory, it is added to the finished list and the machine construction is finished. The scan continues with the new machine. Otherwise new variables/operators are being added to the same machine. The process continues until the end of the postfix string is reached. If all machines are in the finished list, the algorithm terminates. For the given postfix string, the process is shown in Figure 6.11.

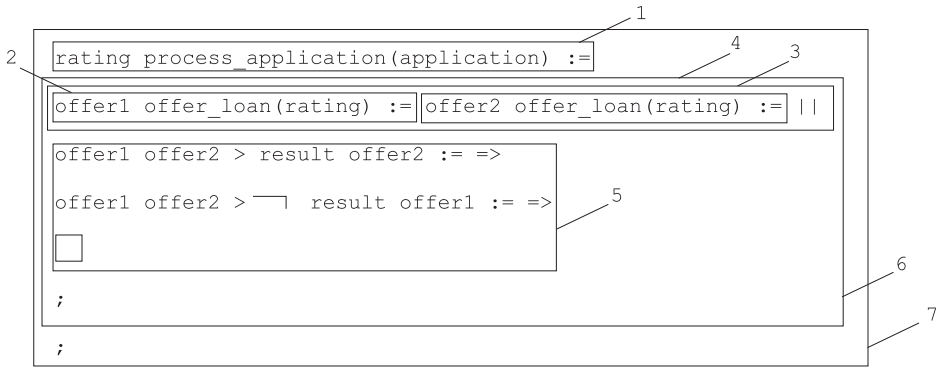

Figure 6.11: Postfix String Scan

In step one, variables `rating` and `process_application(application)` are connected with assignment operator. Suppose that the following abstract machine is in the finished list, that is, it exists in a directory:

```
MACHINE machine_1
VARIABLES application:IN, rating:OUT, process_application
SETS App, Rating
INVARIANT process_application ∈ App → Rating ∧ application ∈ App
∧ rating ∈ Rating
OPERATION rating <- op_1 (application)
PRE THEN rating := process_application(application)
END
```

Note that clause `PRE rating > 0` from the original machine is not included, since `rating` is the output parameter for which pre-conditions cannot be defined. Since the previous part of the string exists in a directory, in

steps two and three, variables `offer1` and `offer_loan(rating)`, as well as `offer2` and `offer_loan(rating)` are connected using assignment operator. Suppose that the following two machines are also in the finished list:

```
MACHINE machine_2
VARIABLES rating:IN, offer1:  OUT, offer_loan
SETS Rating, Off
INVARIANT rating > 0 ∧ offer1 ∈ Off ∧ offer_loan ∈ Rating → Off
OPERATION offer1 <- op_2(rating)
PRE rating > 0
THEN offer1 := offer_loan(rating)
END
```

```
MACHINE machine_3
VARIABLES rating:IN, offer2:  OUT, offer_loan
SETS Rating, Off
INVARIANT rating > 0 ∧ offer2 ∈ Off ∧ offer_loan ∈ Rating → Off
OPERATION offer2 <- op_3(rating)
PRE rating > 0
THEN offer2 := offer_loan(rating)
END
```

The postfix string is then scanned further. The next symbol in step four is parallel composition operator that is applied to `machine_2` and `machine_3`. In step five, the first scanned expression yields the following machine:

```
MACHINE machine_4
VARIABLES offer1:IN, offer2:IN, result:OUT, wcet
SETS Off
INVARIANT offer1 ∈ Off ∧ offer2 ∈ Off ∧ result ∈ Off
∧ wcet ∈ Off → Time
OPERATION result <- op_4(offer1, offer2)
PRE offer1 > 0 ∧ offer2 > 0 ∧ wcet(offer1) > 86400
∧ wcet(offer2) > 86400
THEN (offer1 > offer2) ⟹ result := offer2
END
```

Suppose, however, that this machine does not exist in the finished list. There is no service in a directory that can perform a comparison of two arguments of the type `Off` and return one that is bigger. This is the key point in the decomposition algorithm where a decision has to be made whether to proceed further with postfix string or to go back. If a decision to go back is taken, the previous compositions have to be decomposed in a backtracking process. We would effectively try to incorporate `machine_4` in `machine_3`

or machine_2. The other solution is to proceed forward and to try to incorporate the next part of the postfix string in machine_4. The approach we adopt is that we move one time in each direction until a service that exists is reached. Therefore, in the continuation of step five, machine_4 is modified by scanning postfix string until the next operator is reached:

```
MACHINE machine_4
VARIABLES offer1:IN, offer2:IN, result:OUT, wcet
SETS Off, Time
INVARIANT offer1 ∈ Off ∧ offer2 ∈ Off ∧ result ∈ Off
∧ wcet ∈ Off → Time
OPERATION result <- op_4(offer1, offer2)
PRE offer1 > 0 ∧ offer2 > 0 ∧ wcet(offer1) > 86400
∧ wcet(offer2) > 86400
THEN (offer1 > offer2) ⟹ result := offer2 □
 ¬ (offer1 > offer2) ⟹ result := offer1
END
```

Supposing that this machine exists, we move to step six in which we connect machine_4 to parallel composition of machine_3 and machine_2 using sequence operator. Finally, in step seven machine_1 is sequentially composed to the already built composition. The result is (Figure 6.12):

$$\texttt{machine\_1} \rhd (\texttt{machine\_2} || \texttt{machine\_3}) \rhd \texttt{machine\_4}$$

In case that machine_4 did not exist in a directory, decomposition algorithm would try to backtrack and incorporate all scanned but unassigned variables into the last valid construction, that is, into parallel composition of machine_2 and machine_3. The result of this step is:

```
MACHINE machine_5
variables rating:IN, offer1, offer2, result:OUT, offer_loan, wcet
SETS Rating, Off, Time
INVARIANT offer_loan ∈ Rating → Off ∧ rating ∈ Rating ∧ offer1 ∈ Off
∧ offer2 ∈ Off ∧ result ∈ Off
OPERATION result <- op_5(rating)
PRE rating > 0 ∧ offer1 >0 ∧ wcet(offer1) < 86400 ∧
offer2 > 0 ∧ wcet(offer2) < 86400
THEN [ ( offer1 := offer_loan(rating) || offer2:=offer_loan(rating) );
 ( (offer1 > offer2) ⟹ result := offer2 □
 ¬ (offer1 > offer2) ⟹ result := offer1 ) ]
END
```

Now, if this machine exists in a directory, decomposition result would be:

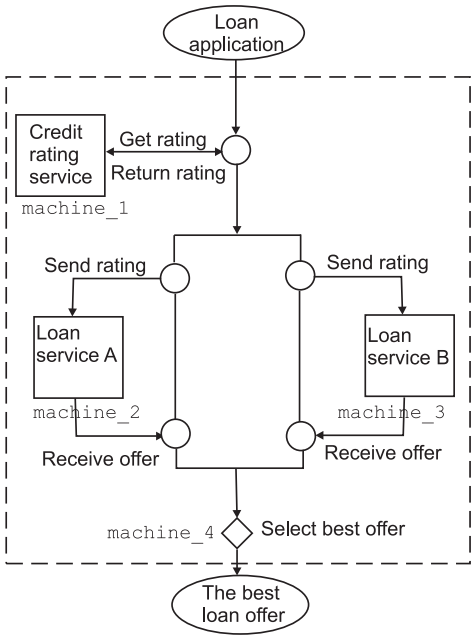

Figure 6.12: Loan Application Composition

$$\texttt{machine\_1} \triangleright \texttt{machine\_5}$$

If not, a process of postfix conversion is applied again, this time to `machine_5`, which finally yields that:

$$\texttt{machine\_5} \equiv \texttt{machine\_2}||_{\texttt{offer1>offer2}}\texttt{machine\_3}$$

and therefore final decomposition is:

$$\texttt{machine\_1} \triangleright (\texttt{machine\_2}||_{\texttt{offer1>offer2}}\texttt{machine\_3})$$

Let us give the complete algorithm. Let $M(O, C, S)$ be the target service with operation body O, clauses C and state variables S:

$PF = \oslash$
$FINISHED = \oslash$
$elem = \oslash$
$CL = \oslash$
$PF = convert(O)$
foreach s in S

foreach c in C

if s in c $\quad CL = CL \cup \{s, c\}$

while ($elem$!= END)

$elem = scan(PF)$

$switch(elem)$

case($variable$) : $elem = elem \cup scan(PF)$

case($operator$)

if ($correct(elem, CL)$)

if ($exists(elem, CL)$)

$FINISHED = elem$

$elem = \oslash$

$elem = scan(PS)$

$if!(exists(elem, CL)$

$elem = scan(FINISHED)$

if $elem = \oslash$ exit
else fail

Two major problems of decomposition are:

- Incorrect abstract machines: what is the behavior of the algorithm when abstract machine scanned in a postfix string is not correct?
- Algorithm missing decomposition path: what happens when abstract machine scanned in a postfix string does not exist in finished (list) directory, and moving forward/backward does not produce an existing machine?

In both cases decomposition algorithm cannot guarantee that a solution will be found. This is the consequence of a very simple control strategy that is adopted for both cases. When either a machine is incorrect or does not exist in a directory, first a forward scan is performed, and in case that also does not result in a valid/existing machine, solution is backtracked in a backward scan. Instead of providing more complex control strategies that could achieve higher hit probability, a hybrid bidirectional search mechanism is developed.

6.8 Hybrid Bidirectional Automatic Composition

The last automatic composition mechanism that will be described is hybrid bidirectional search. The basic idea is to use advantages of both forward and backward search in order to eliminate two problems: complexity of basic heuristic search and missing solution path in decomposition. Bidirectional search simultaneously performs forward and backward search thus creating two search paths. The search is over when (if) the two paths meet, and the solution is constructed by merging them. Forward search can be performed using any of the methodologies presented so far: basic heuristic search, probabilistic search, or search by learning. Backward search is performed by decomposing target abstract machines. There are three ways to perform bidirectional search:

- sequential control strategy
- depth specification
- means-ends analysis

When sequential control strategy is employed to steer direct bidirectional search, steps in both direction are made sequentially. After each step, current states are compared. If the current state of backward search is a subset of the current state of forward search, the algorithm terminates. This approach is simple to implement as it does not require any additional control protocol. However it suffers from one problem: search paths may miss each other, completely or partially (Figure 6.13).

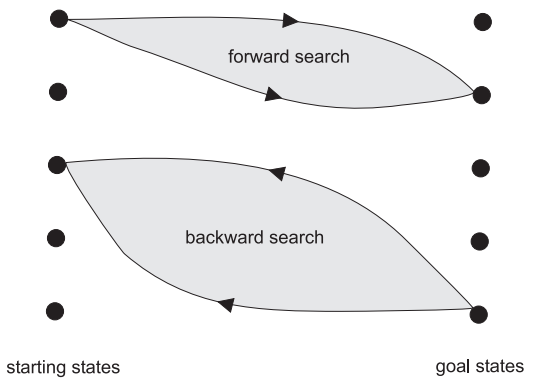

Figure 6.13: Bidirectional Search Problem

In the worst case of successful execution, this kind of search will perform complete forward search, and also spend additional overhead for unsuccessful backward search. That means that it will last longer than the forward search only. Therefore, in the second approach, depth of both directions can be specified as an input parameter. For example, backward search can be allowed to progress for only one level, thus making it easier to satisfy several "smaller" target machines with a subsequent forward search. Similarly, forward search can be allowed to progress to a certain level thus decreasing the number of goal states for subsequent decomposition. In this case, bidirectional search is not performed concurrently in both directions. Rather, first one direction is explored to a certain level, in order to make the search in the opposite direction easier and/or more effective. Bidirectional search (without special control strategy and with both depths set to one) for the loan application example from the previous section is shown in Figure 6.14.

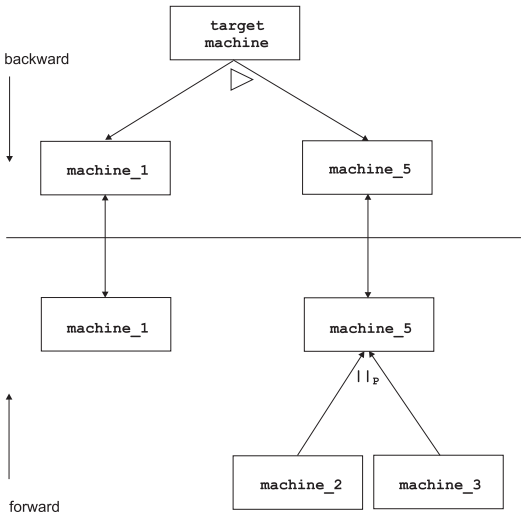

Figure 6.14: Bidirectional Search Example

There are still two open issues: which direction to favor (forward or backward) and up to which depth to limit the auxiliary search? Both questions can be answered by using means-ends analysis [40, 124]. It is a methodology that detects differences between current and goal state and tries to make a move in a state space that will reduce this difference. This method is modified to allow for a decision in which direction to move (forward or backward) and for how many steps. The algorithm proceeds like this:

- Until the current state of backward search is not a subset of the current state of the forward search, or difference table offers no more options,

do the following:

- Calculate the difference between two current states
- Use a distance function and a difference table to determine whether to execute a forward or a backward move
- Update current states accordingly

- If goal is achieved, the algorithm terminates successfully, otherwise it fails.

During search process, pre-conditions and post-conditions are evaluated in order to determine whether forward of backward search should be performed. In forward search, post-conditions of available services are matched against pre-conditions of target services. In backward search, pre-conditions of target services are matched to post-conditions of available services. Matching is performed using a difference table that describes which operation (direction) is appropriate to reduce the difference between the current and the goal state. Matching is best described using an example: suppose we want to print a Word document and only a printer that can print postscript with embedded fonts is available. Available are also the following services: converter from Word to pdf format, converter from pdf to postscript format and service that can generate pdf file with embedded fonts. Abstract machines describing these services are:

```
MACHINE PrintPS
SETS Type={Word,PDF,PS}, Paper, Document
VARIABLES doc, print, fonts, type, result
INVARIANT print ∈ Document → Paper ∧
fonts ∈ Doc → {Embedded,NotEmbedded} ∧ type ∈ Document → Type
OPERATION result <- printPS(doc)
PRE doc ∈ Document ∧ type(doc) = PS ∧ fonts(doc)=Embedded
THEN result:= print(doc) ∧ result ∈ Paper
END

MACHINE Fonts2PDF
SETS Type={Word,PDF,PS}, Document
VARIABLES type, doc, fonts
INVARIANT type ∈ Document → Type
∧ fonts ∈ Document → {Embedded,NotEmbedded}
OPERATION doc <- fonts2PDF(doc)
PRE doc ∈ Document ∧ type(doc) = PDF
THEN type(doc) = PDF ∧ fonts(doc) = Embedded
END

MACHINE PDF2PS
```

```
SETS Type={Word,PDF,PS}, Document
VARIABLES doc, type
INVARIANT type ∈ Document → Type
OPERATION doc <- pDF2PS(doc)
PRE doc ∈ Document ∧ type(doc) = PDF
THEN type(doc) = PS
END

MACHINE Word2PDF
SETS Type={Word,PDF,PS}, Document
VARIABLES doc, type
INVARIANT type ∈ Document → Type
OPERATION doc <- word2PDF(doc)
PRE doc ∈ Document ∧ type(doc) = Word
THEN type(doc) = PDF
END
```

Difference table is shown in Figure 6.15. Sometimes there may be more than one operator that can reduce a given difference (word2pdf, pdf2ps, fonts2pdf), but also one operator may be able to reduce more than one difference (fonts2pdf). Assuming that target machine is given below, the search proceeds like in Figure 6.16:

```
MACHINE print
SETS Type={Word,PDF,PS}, Document, Paper
VARIABLES doc, type, print, result
INVARIANT type ∈ Document → Type ∧ print ∈ Document → Paper
PRE doc ∈ Document ∧ type(doc) = Word
THEN result := print(doc) ∧ result ∈ Paper
END
```

	word2pdf	pdf2ps	fonts2pdf	printps
print				√
convert	√	√	√	
embedd			√	

Figure 6.15: Difference Table

In the first step, backward search is performed and from difference table operation `printPS` is selected. However, service `PrintPS` is not equivalent to the goal state, since it accepts only Postscript documents with embedded fonts. Further decomposition yields no result because there is no way either to transfer Word file directly to Postscript or to embed fonts into a Postscript

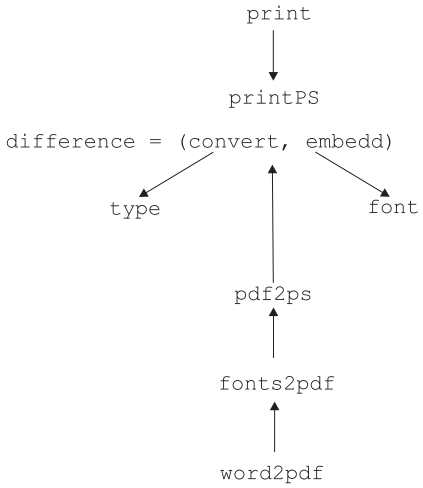

Figure 6.16: Means-Ends Bidirectional Search

document. Therefore, at this point a difference is determined and forward search is attempted. The difference is type of input parameters and whether fonts are embedded within the document. Difference table suggests using operation `fonts2pdf` to reduce font difference. By examining pre-conditions we see that this operation can be performed for pdf documents only. That means that this difference cannot be reduced at this point. Difference table suggests three operations for reducing type differences: `word2pdf`, `pdf2ps` and again `fonts2pdf`. Pre-conditions determine that only `word2pdf` operation can reduce the difference. At this point we have a pdf document without embedded fonts. Now operation `fonts2pdf` can be applied thus eliminating one difference (font embedding). After that `pdf2ps` is applied to reduce the last remaining difference (document type). The two searches meet and the algorithm terminates.

Finally, the complete algorithm for the case when means-ends analysis control strategy is used is given. Let $G(PRE, POST)$ be the target service with pre-conditions and post-conditions and $S(S_1(PRE_1, POST_1), ..., S_m(PRE_m, POST_m))$ be the set of available (atomic) services:

$BACKWARD = G$
$FORWARD = \oslash$
while(!($BACKWARD \subset FORWARD$))

$distance = \delta(BACKWARD, FORWARD)$
$operation = lookup(difference\_table, distance)$

$switch(operation)$

$case(forward)$:

$FORWARD = FORWARD \cup compose(operation)$

$case(backward)$:

$BACKWARD = BACKWARD \cup decompose(operation)$

It is implicitly assumed that function *lookup* performs matching of pre- and post-conditions when doing difference table lookup, as well as that pre- and post-conditions are updated in $FORWARD$ and $BACKWARD$.

6.9 Analysis and Comparison

Since automatic service composition was modeled as a search problem, the analysis of developed algorithms will be performed according to this fact. There are two fundamental approaches for judging the quality of a search algorithm: determining how fast it executes (time and space complexity) or how good are the answers (solutions) that it produces. In this case, the latter metric is not relevant, since absolute solution is either achieved or not: at the moment human interaction with suboptimal solution is not taken into account. Therefore, we will concentrate on the complexity and execution speed.

Let us first examine the basic heuristic search algorithm. The complexity of one algorithm cycle (search for candidate nodes with the smallest δ and their expansion) will be observed as a function of the number of nodes n in the `OPEN` list. The complexity is proportional to:

$$n \cdot (n + n \cdot n) = n^3 + n^2$$

Therefore, complexity of one algorithm cycle is $O(n^3)$, where n is the current number of the nodes in the `OPEN` list. This number, however, changes with every algorithm cycle, therefore a more important question is how this number changes in every algorithm cycle and how many cycles does the algorithm require in order to find a solution. Let m be the number of atomic services, o the number of composition operators and s the number of algorithm cycles. The number of generated nodes in each algorithm cycle is then given by:

$$m \cdot (o + 1)^s$$

Clearly, it follows that algorithm has an exponential complexity. However, s is the function of F, meaning that exponential growth of generated nodes is

the upper complexity bound, or the worst case complexity, when all generated nodes fall below the futility value. Analytical dependency between s and F is very difficult to express in the general case as shown in [55] for the A* algorithm. Instead, we will discuss the possible ways to build value F in order to reduce the complexity.

There are two ways to determine F: to define the maximum value of the distance function δ for which it is still feasible to examine the current path (this approach is used in Section 6.4) or to use some other metric. If distance function is used as futility metric, futility can be either chosen as the distance function value of nodes without successors, or the value can be determined empirically (application specific). Other metric applicable for F is to calculate the number of generated state variables for each node. If this number is greater than the number of state variables for the goal, the current node is moved to the `LIMIT` list, as subsequent compositions can only increase the number of state variables. This means that machines with more states than the goal machine are not accepted as solutions, which is compliant to definition of distance function. This is somewhat problematic as subsets are not taken into account: services that can satisfy search criteria and offer extra functionality will not be accepted as solutions. The problem in such cases is, however, easily mitigated by comparing operations of abstract machines that offer extra functionalities (more web methods) separately. Apart from determining futility value, other methods can be used to improve search performance, such as introducing execution time to limit the search temporally. This clearly makes sense in some applications where time limit can be naturally determined (e.g., bank transaction involving currency conversion, credit card verification and payment should not take longer than 1 minute). The number of generated nodes can also terminate the search, as storage capacity is not infinite, and all generated nodes must be stored, even those that have crossed the futility value (`LIMIT`) or have already been expanded (`CLOSED`). This is required because of the equivalence comparison when new nodes are generated in order to detect equivalent and redundant paths.

In the worst case, the other two forward search strategies fall back to the basic heuristic search, thus having the same upper bound complexity. On the other hand, worst case complexity of backwards search is $O(n^2)$, assuming very simple conflict control strategy that has been described. Finally, the complexity of the bidirectional search is very difficult to determine analytically, as it depends on the direction depth. It can vary from $O(n^3)$ to $O(n^s)$. Clearly, the analytical analysis shows that the preferred method for automatic composition should be decomposition (backwards search), at least judging by the worst-case complexity. However, in many heuristic approaches, average-case complexity is much more important. Since we expect

that these heuristics will hopefully never execute in their upper complexity bounds, it is both worth and relevant to examine their average performance through experiments.

The experiment was performed using twenty basic operations from the well knows holiday booking scenario exposed through Web Services (e.g., reserve a flight, reserve hotel room, rent a car, charge credit card, etc.), and the task was to solve ten different composite requests. The results are, therefore, averaged over ten algorithm runs for different goals. Non-functional properties that were modeled included security (access right to different services), dependability (some services could run inside a single transaction, others not), and timeliness (execution time was defined for several operations). The results given in Figure 6.17 relate to physical execution speed (the absolute time in seconds required to find a solution) and to the number of compositions (expanded nodes) in each case.

On the average, the fastest algorithm is bidirectional search with depth specification, followed by decomposition and bidirectional search using difference table which execute with almost identical speed (Figure 6.18). The reason that depth specification is clearly superior to means-ends analysis in terms of speed is that means-ends analysis requires costly operations of difference table lookup in every step. Probabilistic approach with adaptive probabilities leads the second group, closely followed by surprisingly fast basic heuristic search. The reason why basic heuristic search outperforms probabilistic and learning-based search lies in the simplicity of calculating heuristic function and the fact that it does not require maintenance of complex structures (cooperation graph, classification). The slowest approach is the learning-based approach, which is almost four times slower compared to the fastest solution, because of the complex data structures that it maintains.

	average execution time [s]	number of compositions [expanded nodes]
heuristic	2.59	412
fixed probabilities	3.24	350
adaptive probabilities	2.16	297
learning	4.03	308
decomposition	1.57	284
bidir (depth-spec)	1.07	302
bidir (means-ends)	1.69	202

Figure 6.17: Performance Comparison of Automatic Search Algorithms

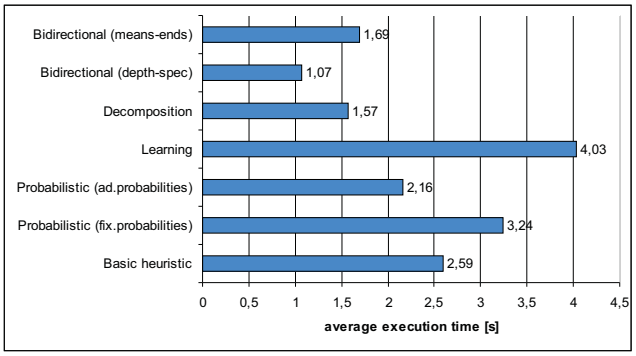

Figure 6.18: Automatic Composition Average Execution Time

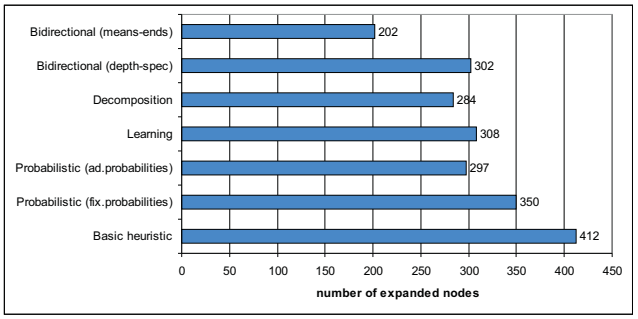

Figure 6.19: Automatic Composition Average Number of Compositions

The other metrics, however, presents somewhat different picture (Figure 6.19). The best algorithm in terms of number of compositions performed (number of expanded nodes) is bidirectional search using means-ends analysis (difference table). The fastest algorithm, bidirectional search with depth specification, is only the third here, since even decomposition offers a slightly better path through the state space. Probabilistic search with adaptive probabilities and learning follow close, while by far the worst performance is offered by basic heuristic search which executes with over $O(n^2)$ average complexity in this case.

It can be concluded that there is no single metric that can be used for comparison of developed methods. It has been shown, on the example of best performing algorithms in both categories, that fast movement through the state space requires significant computing overhead in maintaining helper data structures. It is also visible that relatively simple basic heuristic search executes very fast (easy calculation of heuristic function) despite the worst

average execution complexity. The two design criteria (execution time and minimizing number of expanded nodes) are clearly conflicting. Based on the presented results, it would be wrong to conclude that execution time is the only relevant metrics, since, in this experiment all services were executing on the same machine and node expansion cost may not have been realistic enough (e.g., network latency penalty was not included). However, the results clearly show that bidirectional search and decomposition offer the best compromise. Even more important result is that the average execution complexity of all presented heuristics is below $O(n^2)$ in this example, which is a favorable feasibility indication. All the experiments were performed when solution was existing, that is, when given trip plan (goal service) could be satisfied using available operations (atomic services). When the solution does not exist, all algorithms execute in their upper complexity bound before reporting that solution does not exist, except when they are time limited (e.g., timeout). In this case the superior solution is decomposition ($O(n^2)$), but as already shown, the penalty is that it does not guarantee a solution. Apart from decomposition, all other algorithms will execute with exponential upper complexity bound, with high probability that bidirectional search will report non-existing solution in polynomial time, if difference table or depth specification favor backward (polynomial) instead of forward (exponential) direction.

6.10 Related Approaches

In this section a discussion of the existing related approaches to automatic service composition is presented. Two generic strategies will be described, namely constraint satisfaction/planning and ontology-based logic reasoning (deductive databases), together with comparison with the methods we have proposed.

In [3, 169] a method is presented that reduces service composition to a constraint satisfaction problem. The main entity is an abstract process which contains abstract services. An abstract service is a placeholder for a set of physical (real) services that match the abstract service template, effectively competing for its place. Competition is based on the idea of automated service discovery [30, 127]. Automated discovery is performed using user defined requirements and produces set of candidate services. After discovery, candidate services are selected on the basis of process and business constraints.

The main stages of creating dynamic process are development, annotation, discovery, composition and execution. Different semantics can be used:

data, functional and quality of service. The part that is most relevant for automatic composition deals with design of abstract processes. It involves the following steps (Figure 6.20):

- Creation of desired flow using control flow constructs provided by BPEL.
- Annotating BPEL flow using templates that express service properties.
- Specifying constraints that will be used for optimization.

BPEL annotation is performed using different ontologies. Partner services are represented as annotated abstract services. Then a search on extended UDDI is performed, and for each template a set of matching services is identified. In the process of optimization, constraints are evaluated and candidates are eliminated. Constraint satisfaction can be performed upon service dependencies, querying and cost estimation or process constraints. After candidate services have been filtered and identified, BPEL process is translated into executable form by adding physical addresses of selected partner services to BPEL deployment descriptor and sent to BPEL server for execution.

In [57] a system called Proteus is presented that uses planning techniques for dynamic composition and execution of Web Services. The main feature of Proteus is dynamic composition of plans that integrate Web Services. Besides planning, Proteus offers plan execution and monitoring.

The system behavior is very similar to constraint satisfaction. Service description is annotated with additional expressions using WS-Inspection [15]. The annotated plan is submitted to a search engine that tries to find adequate services at run-time and substitute them in the plan. An integration plan is generated that binds identified services into the requested plan and this plan is then executed.

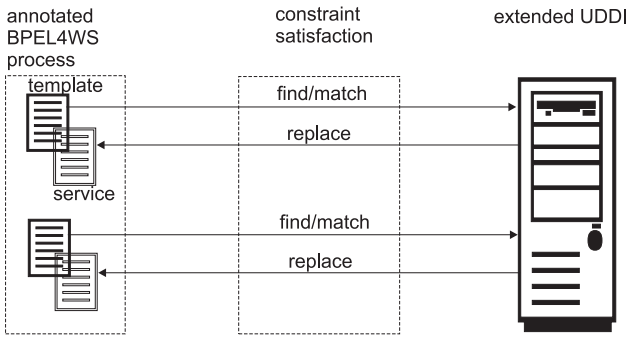

Figure 6.20: Constraint Satisfaction-based Automatic Composition

In [190] another method similar to constraint satisfaction is presented. Composition is modeled using statecharts over which execution paths are constructed. Statechart comprises generic elements called tasks. Execution path is any complete path of tasks from starting state to ending state. Any subset of a set of all possible execution paths for which quality of service properties are evaluated is a potential execution plan. The problem is how to identify the execution plan that corresponds to the user-expressed requirements. If there are n tasks (states) and m candidate Web Services that are identified for each task, the number of all execution plans is m^n which makes this method of automatic composition impractical. Therefore, integer programming [73] is used for selecting an optimal execution plan without the need to generate all possible plans.

Inputs to planning are a set of variables, objective function and set of constraints. Set of variables describe quality of service properties of each task that is being considered. Constraints are user-defined limitations on price, execution duration, execution price, reputation, success rate and availability. Objective function compares current execution plan to the constraints. Both objective function and constraints are linear. In the process of composition value of objective function is maximized or minimized by adjusting the values of variables while enforcing the constraints. The output is the maximum (or minimum) value of the objective function from which the values of the variables can be extracted for this maximum (minimum). The set of variables determines which candidate service instances actually populate tasks during physical execution.

Finally, in [153] an approach to automatic composition based on semantic web is presented. It is based on OWL-S ontologies (already discussed in Chapter 2). More specifically, OWL-S process model is used to develop a desired composition by creating a composite process comprising choreographed atomic processes. After composite OWL-S process is created, a search is performed in order to find the best matching services that can replace atomic processes (abstract service placeholders).

Automatic composition has two main components: composer and inference engine. The inference engine is essentially a directory that has the capability to find matching services that best fit specified abstract processes. It is designed as a knowledge base using Prolog. The composer is the interactive part of the system. It enables user to create a workflow of services and it also presents all available choices to the user at every step. That means that despite knowledge base, composition has to be performed (partly) manually. At every step, functional and non-functional properties of participating services are matched, and some candidates are rejected. This process can also be assisted by a human operator.

After a desired composition has been found, that is, after all abstract processes from OWL-S process model have been substituted by real services from directory, the entire composite process is stored in a directory from which it can be invoked (executed).

The second group of approaches for automatic service composition is based on reasoning performed upon deductive databases. Rules describing the system (onotolgies) and descriptions of available services are stored in a database. The system is presented with a query describing current (starting) state and the goals. It is expected that the reasoning engine will be able to compute a state transition from the current to the goal state using rules and available services. Reasoning mechanisms are dependant on the formalism used for rule and service description, and several relevant solutions will be described.

MyGrid project [149] uses federated UDDI directories annotated with RDF to provide semantic description. The language used for onotology description is OWL-S. Based on the given goal and available semantically-enriched service advertisements, this approach uses description logic [11] to perform reasoning in order to match request with available resources (services). Reasoning operations that can be performed are instance checking, subsumption reasoning, etc.

InfoSleuth [45] is an agent-based system that uses Open Knowledge Base Connectivity (OKBC) to represent and store ontologies. Several agents are available in the system, that are used to match the user's goal: user agent, ontology agent, broker agent, resource agent, data analysis agent, task execution agent and monitor agent. Agents communicate using Knowledge Query and Manipulation Language (KQML). When user submits goal task, deductive database storing semantic description of available services and rules is queried using deductive database language (DDL++) thus semantically checking if the query matches available advertisements.

Language for Advertisement and Request for Knowledge Sharing [160] is an agent-based approach for agent (service) matching. There are three agent categories: service providers, service requesters and middle agents. The middle agent matches the query against the advertised providers capabilities. In this process the middle agent uses advertisement database and partial global ontology. A frame-based language is used to describe queries (goals) and advertisements. Ontologies are used to describe the meanings used in queries and advertisements. The language used for ontology description is ITL. The reasoner offers the following matchings: subsumption reasoning (context and profile matching), subtype inference rules (signature matching) and subsumption reasoning for Horn clauses (constraint matching).

The Web Service Modeling Ontology (WSMO) [146, 180] is a framework

for automated Web Service discovery, selection, composition, execution and monitoring. Ontologies in WSMO are described using one of the Web Service Modeling Language variants [181]: WSML-Core, WSML-DL, WSML-Flight, WSML-Rule and WSML-Full. They are based on description logic, first-order logic, logic programming and description logic programming. Several reasoners (e.g., WSML-Rule and WSML-DL) have been already developed, while other native WSML reasoners are work in progress. Based on descriptions of user requirements and service advertisements, reasoners perform matching with the help of information found in ontologies.

Finally, there are numerous approaches to use reasoning upon OWL-S ontologies to perform automatic composition [90, 126]. They all use description logic formalism, and are able to compute the following degrees of matching: exact match, subsumes matching, intersection matching and disjoint matching (failed matching). Naturally, OWL-S specifications are used to describe both user requests (goals) and available service descriptions. Methods like subsumption reasoning and instance checking are then performed for submitted goals in order to identify services (capabilities) that can fulfill them.

From the solution we proposed and from the related solutions presented in this section, it can be seen that there are two fundamentally different ways to handle automatic service composition:

- To start with the pre-defined composition described in a generic manner (empty service placeholders connected using execution logic) and to perform 1-1 search in a directory to replace every generic element of a composition with a real service.

- To describe a set of goals and try to achieve them by building the whole composite process from scratch, without prejudicing neither the number of services nor the logic that is used to connect them.

Clearly, constraint satisfaction, planning and integer programming belong to the first approach. They all provide methodology to describe pre-designed service choreography (empty composition skeleton) with placeholders that are to be filled with actual (real) services. They thus reduce the problem of automatic composition to the problem of finding adequate replacement for every abstract element of the pre-defined composition. It is not possible, for example, to replace two abstract activities with one concrete service that matches the sum of two constraints instead of each of these constraints alone. We feel that service-oriented application designer should not think in terms of pre-defined compositions, but in terms of the problem that is to be solved. Our approach therefore does not require that composition should be predefined. Target abstract machine specifies properties of the problem (goal)

itself, and not the way to achieve it. It does not prejudice composition process by requiring that certain services should be composed in the given manner. Identification of composition patterns and candidate services is left to the automatic composition algorithm based on the target abstract machine.

Let us consider an example from Section 6.7, modeling loan flow composite service. For constraint satisfaction and planning solutions presented in this section, an expert human knowledge would have to be used not only to describe problem properties, but also to make initial composite process layout in terms of services and their compositions. In this case, it would mean that input to the system would be predefined BPEL composition without deployment descriptor, that is, without binding to existing services. This would require that assumptions should be made about the number of services required to solve the problem, their properties and ways to connect them. The designer would have to specify abstract credit rating service, abstract bank services and logic that controls their execution. In the subsequent process of directory search, each `partnerLink` element from the above will be instantiated with a real service, admittedly with checking whether it preserves overall composition properties. On the other side, our approach does not require human knowledge in the area of service placeholder selection and their connection. It is enough that composite process is described using either B notation or CDL, and both number of services and their connections (composition logic) will be *discovered* during automatic composition. That way, many solutions can be covered, contrary to the approaches that try to fit existing services in a single, predefined composite logic. Automated planning methodologies other than constraint satisfaction (e.g., deductive planing, resource scheduling, task decomposition, propositional satisfiability, or model checking) could eliminate the downside of constraint satisfaction approaches as they allow for the goal-based automatic plan generation [56]. However, there are no current proposals to use these techniques for automatic Web Service composition.

The problem specification domain is in that way decoupled from the solution specification domain. By specifying target abstract machine, we do not think about *how* to solve a problem, but *what* and under which *conditions* we need to achieve. We see this as clear advantage compared to other automatic composition concepts that require that entire solution should be premeditated in advance.

Reasoning-based approaches admittedly belong to the second group, as they all allow specification of goals and compute the solution out of available services, without requiring that user either knows properties of existing services or predefines abstract composition skeleton. However, the following problems can be identified in reasoning-based approaches, which are elimi-

nated by our approach of treating automatic composition as a search problem:

- *Speed.* Search algorithms are simpler, offer the lower complexity and execute faster when compared to the logic reasoners.
- *Manual Composition.* The proposed approaches are not well-suited for manual composition, which limits their usage in the application development scenarios, where user (designer) requires greater, manual control over composition partners but still needs powerful verification mechanisms.
- *Deductive databases limitations.* The existing deductive databases are limited technologically, as they do not support transaction handling, load balancing and similar properties that are required in a highly dynamic and stressed environment.
- *Ontology Dependence.* All reasoners depend on the ontology quality. If an ontology is partial, ambiguous or in some other way incomplete, reasoner cannot guarantee solution, even if it exists. Furthermore, current reasoners cannot load ontologies on demand, meaning that all necessary ontologies must be known in advance and preloaded.

For the reasons explained, it is our belief that the novel approach of treating automatic service composition as a search problem presents a research direction that should be further investigated [1].

[1] It should be noted here that *after* we presented the idea of treating automatic service composition as a search problem in [113], another approach to automatic composition using modified AND-OR graphs has been proposed in [91]

Chapter 7

Composition Server Implementation

In this chapter an implementation of the contract-based composition server for Web Services is presented. Composition server is based on the model previously described and allows for publication of services to a directory, search, and composition (manual via graphical user interface and automatic by specifying target service).

7.1 System Overview

The composition server comprises four main parts:

- client application
- administrative services (middle layer)
- service directory (database)
- one or more application servers/containers hosting deployed services

Client application supports two roles: administrator and client. This application is used by server administrators to deploy and configure services (`admin` role), and by clients (users) to search and compose services (`user` role). Both roles do not access directory or application server(s) directly but via the middle layer.

The middle layer holds basic server functionalities. It offers publication of new services, modification or removal of existing services, search for published services, service composition (manual or automatic), and invocation of single or composite services. The middle layer connects to the underlying database

(directory), as well as to the application servers hosting deployed services. It is also responsible for securing and maintaining infrastructure requirements, such as transaction management, exception handling and state management. All functionalities of the middle layer are exposed to the interested clients as Web Services themselves.

Service directory is a relational database that contains description of published services. Descriptions of both atomic services and their compositions can be stored here. Finally, deployed services are hosted in their own, independent containers that are being addressed from the middle layer.

The composition server is realized entirely in Java and Java-related technologies (Swing, Java Architecture for XML Binding, Java API for XML-based Remote Procedure Calls).

7.2 System Model

A composable service-oriented architecture offers a novel approach for the methodologies and concepts that are used to develop and consume enterprise service-based distributed applications. The major roles (actors) involved in application development and exploitation process are: atomic service developer, atomic service deployer, composable architecture deployer, composite application designer, and application consumer (Figure 7.1).

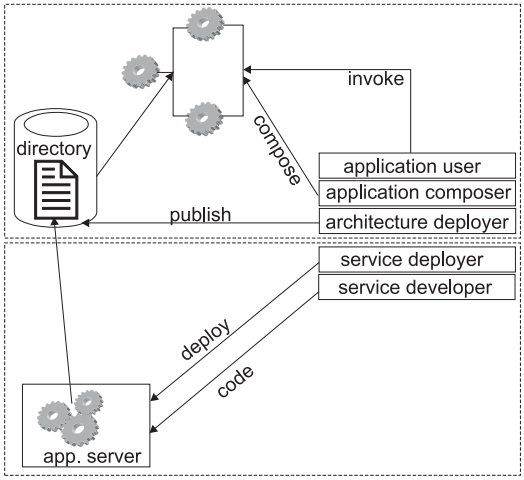

Figure 7.1: Roles in Composable Service Architecture

Atomic service developer designs and codes low-level and fine grained business, data and presentation logic. This task is (almost) application server

independent, since its main focus is on coding service logic, and not on binding the service with the environment in which it will eventually execute. Atomic service deployer wires service with its native execution environment by writing deployments descriptor (e.g., setting transactional support), specifying data access dependent configuration information and finally deploying a live and running service on a specified endpoint to its container. In this step source code can still be augmented or modified due to container-specific requirements. Service deployer is further in charge with ensuring optimal running and execution of atomic services by maintaining their native application servers. Optionally, a service contract is manually added or extracted by any of these two roles. At least semi-formal and standardized documentation must be produced to be used for contract creation by other roles. Composable architecture deployer ensures that all services have well-defined contracts, publishes contracts to directory, defines composition operators, generates additional minimization rules, sets up reputation system and maintains composition server. This role thus controls composition rules and processes, based on the available atomic services that have been independently deployed in their own containers. In case that services already have well-defined contracts, access to the source code is not necessary. Otherwise, architecture deployer can still request code inspection, but only for the purpose of valid contract generation. Code is not changed and/or recompiled by this role. Instead, service description is made public for composite application designers and users. Architecture deployer is also in charge of exchanging service description data with other directories, where this issue is not taken care of by other means (e.g., automatic periodical exchange). Composite application developers access service directory and perform composition of services contained therein in order to build complex service-based applications matching business and operational requirements. This process can be aided by rapid application development tools that connect to the underlying directory and composition server. Compositions can be stored in the directory to be further composed and/or reused. Finally, application users browse directories and locate single and composite services fulfilling their needs. Users are not supposed to perform manual composition, but can require nonexisting services to be composed automatically, on-demand. UML use cases diagrams describing these roles are shown in Figures 7.2 and 7.3.

The composition process is performed in two ways, as shown in the UML activity diagram (Figure 7.4): manually and automatically. When composition request is received, it is processed to determine whether it is a manual request, in which case request already consists of partner services and composition operators. Correctness verification of such a request is performed (con-

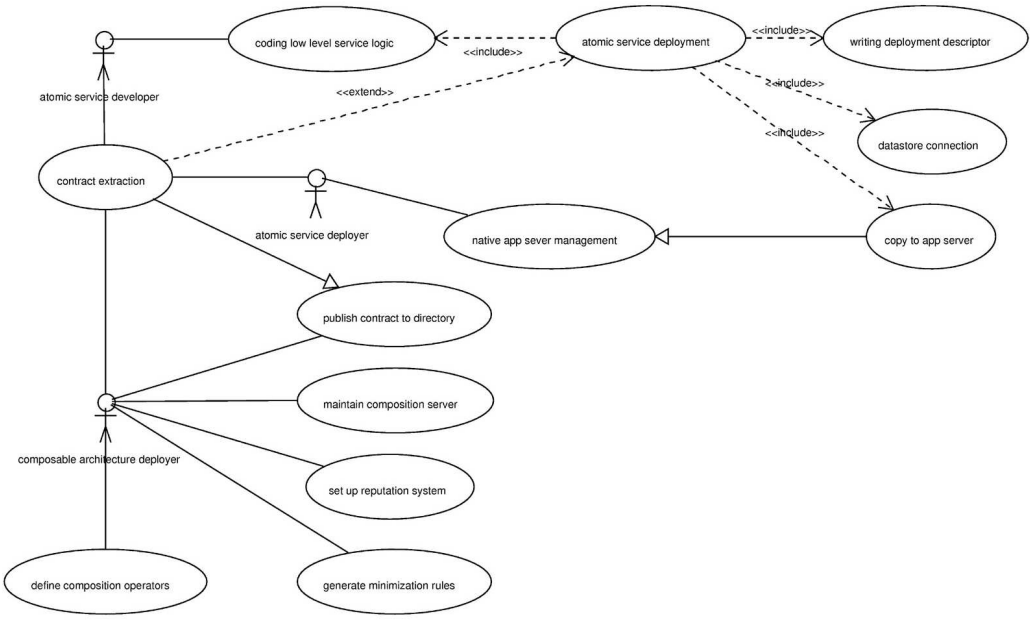

Figure 7.2: Use Cases for Deploying and Maintenance

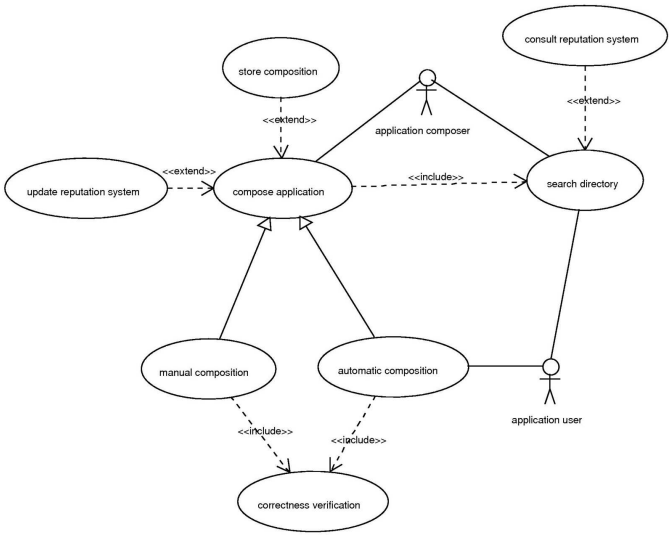

Figure 7.3: Use Cases for Composition

tracts fetched from the directory, composite abstract machine constructed and all steps of verification performed). If composition is not correct, it is

discarded and the system is ready for the new request. In case of automatic composition, the user supplies target service description instead of partner services that he/she wishes to compose. Therefore, automatic selection is performed by executing one or more automatic composition algorithms to determine the most feasible composition that matches the user's request. Verification of correctness is performed upon the results, and if it is passed, the server generates proxies for partner services, both for manual and automatic composition. Proxies are then bound to their implementations, and composition is executed. During the composition, exceptions may be raised. Due to that fact, reputation system is being updated parallel to execution monitoring. In case of exceptions and/or faults, composition is replanned (see Section 7.3.8), alternative composition is generated, its correctness is verified and the process of execution, starting with the proxy generation, is performed. Once execution is completed, results are generated and returned to the client.

The benefits and impact of the proposed architecture to the software life cycle are:

- *Rapid application development.* Application designers and developers benefit from the highly structured way in which application development is realized. Separation of concerns is clearly defined, as well as access to the source code, documentation (contract) and service deployment parameters. Furthermore, designers are not restricted by platform- and language-specific constraints.

- *Software verification and validation.* Composable service architecture introduces application development as formal and structured way of composing atomic services and offers means to verify composition correctness with respect to a variety of properties (security, dependability, timeliness, feasibility, etc). Verification is performed at several levels, including verification of atomic services by developers and verification of the entire composite applications comprising atomic services by application composers. Although the presented framework does not feature validation methodologies as all verification steps are performed in design time, it is possible to imagine contract-based automatic test generation as one viable solution.

- *Maintenance.* Application maintenance is facilitated by clear separation of administrative domains. Service developers and deployers maintain atomic services as well as application servers hosting them. Architecture deployers manage composition server, leaving composite application developers free of administrative tasks, as they concentrate

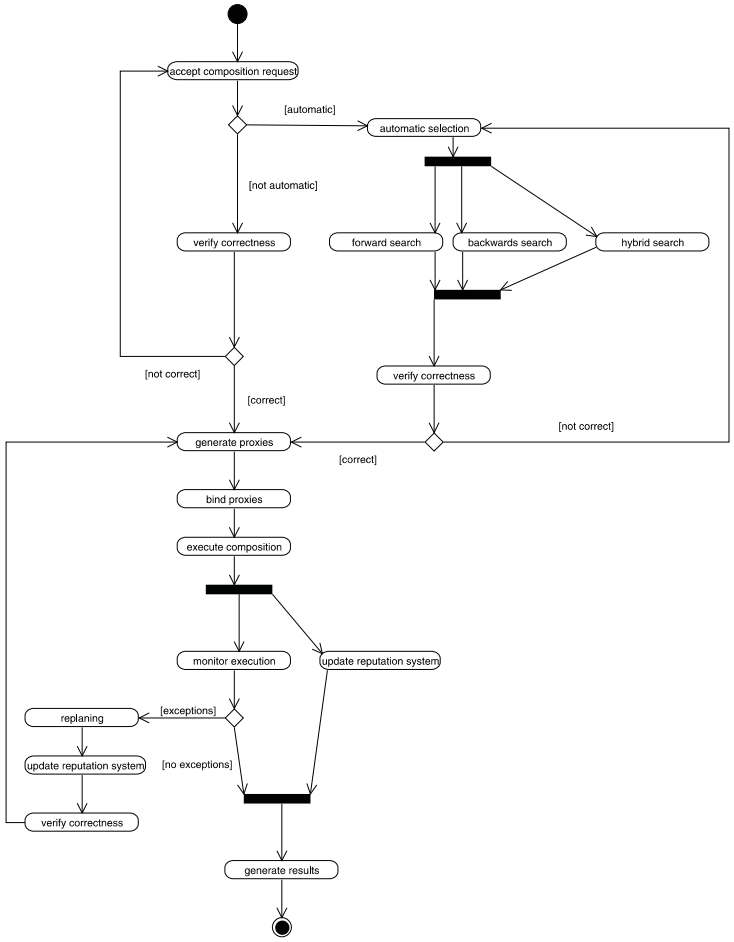

Figure 7.4: Composition Activity Diagram

only on design and application of the composition rules to already managed units (services).

- *Service availability.* Proposed architecture has a strong impact on the service availability, both physical and user-perceived. Physical availability is improved by more efficient verification and management of unit services. Up to now, user-perceived availability has been partially neglected. However, user-perceived service availability is much more important, since it is the true measure of application usability. Simple aggregation of "reliable" services will not produce a "reliable" application. Therefore, roles of architecture deployer and application composer, together with possibilities for verification of composite applications, add another vital layer of availability assessment and pro-

curement.

7.3 System Architecture

In this section, system architecture will be detailed (Figure 7.5), including client application, architecture of the middle layer (XML processing, Web Service communication, composition and proving), connection with a directory, infrastructure management (transactions, exceptions and state management) and finally, how composition server manages execution of a composite service in the presence of failures.

7.3.1 Client Application

The client application supports two roles: `admin` (server administrator) and `user`. User is allowed to search for services based on their properties specified in contracts, to compose and verify services using available composition patterns and to invoke single services or compositions of thereof. Composition is performed using graphical user interface (drag-and-drop) or by typing composition construct manually. For the purpose of automatic composition, target abstract machine is specified or loaded from the text file, and adequate composition is then displayed (if exists) and can be executed. In addition to this, administrator has the option to publish new services to directory, modify and delete existing services and configure server setup (e.g., database location, application servers parameters).

The client part is realized using Swing and is completely decoupled from both underlying database and application servers hosting target Web Services. Instead client application connects to the middle layer from which it obtains all required functionalities as Web Services. Such architectural solution enables easy changes and updates to composition engine, directory structure and application containers without the need to change the client part.

Appendix G shows the main client interface.

7.3.2 The Middle Layer: Administrative Services

Middle layer offers the following operations: publishing new service to directory, modifying and deleting existing service from directory, searching for services, composing new services using existing ones (manually or automatically), invoking single or composed services. In order to achieve these tasks, middle layer communicates with underlying relational database (directory)

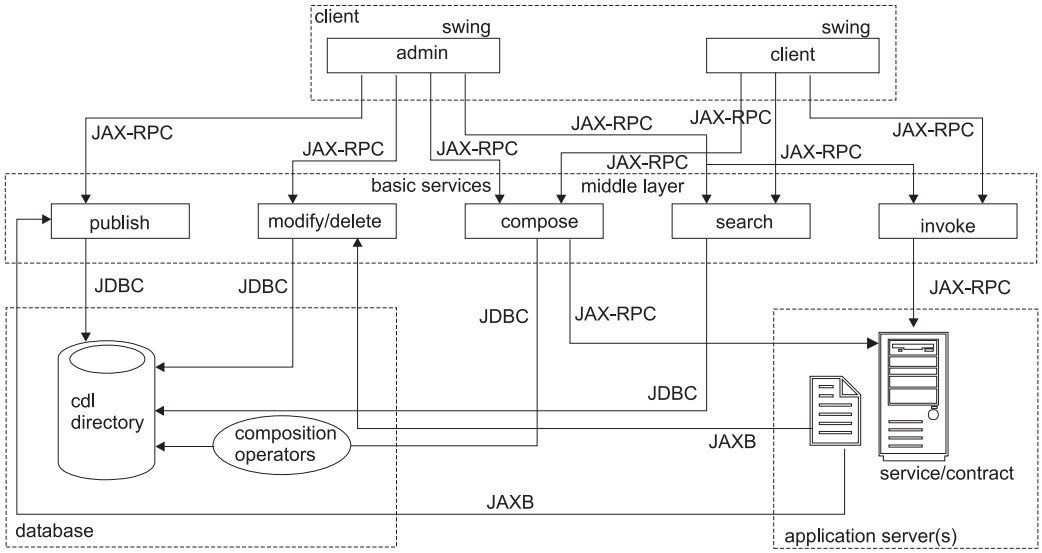

Figure 7.5: Overview of the Composition Server

and application servers hosting target Web Services that users want to invoke and/or compose. Since the entire middle layer also exposes its functionality as Web Services, the communication is realized using Sun's Java Web Services Developer Pack (Sun JWSDP) [105].

We found two technologies provided within JWSDP very useful: Java Architecture for XML Binding (JAXB) and Java API for XML-based Remote Procedure Calls (JAX-RPC). They are essential for understanding how composition engine works and both of them will be described in more details.

XML Processing

Since CDL schema is very large, encompassing more than 50 complex entities, a powerful yet flexible mechanism of translating XML document into Java object representation is needed. The problem can be solved partially using parsers like SAX or DOM, but it would only account for the parsing. JAXB offers a complete solution for transferring XML content into Java object representation and vice versa. JAXB operation is based on three actions: binding XML schema to Java content classes, unmarshalling XML document into content classes and marshalling content classes into XML document. Figure 7.6 shows basic JAXB functions.

JAXB involves two *discrete* groups of actions:

- Generating and compiling JAXB classes from a source schema (CDL)

and building an environment that uses/implements these classes.

- Running the application which unmarshals, processes, validates and marshals XML content (contracts) through JAXB binding framework.

These two steps are usually performed at separate times and in two distinct phases. Typically, in the application development phase JAXB classes are generated and compiled, and a binding implementation is built. This creates an infrastructure that can accommodate actual instances of XML documents. In a deployment phase, generated JAXB classes are used to process XML content in an ongoing "live" setting.

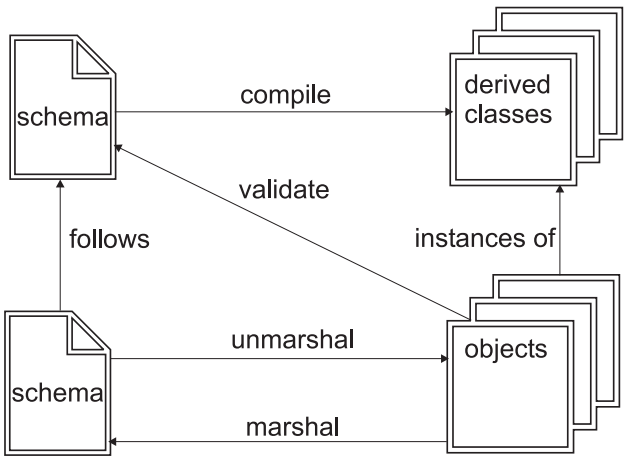

Figure 7.6: Java Architecture for XML Binding (JAXB)

At the beginning CDL schema is compiled with JAXB binding compiler. This action produces a set of hierarchical Java content classes that reflect the contract structure. The class at top of the hierarchy is `Contract`:

```
public interface Contract
    extends javax.xml.bind.Element, test.service.ContractType {}
```

The `Contract` class extends the `ContractType` class which contains actual definition of contract structure. Attributes of the `contract` element are accessed with getter/setter methods operating on primitive Java types, while child elements are processed by fetching their own class. Only one part of the `ContractType` class is shown here:

```
public interface ContractType {
```

```
    java.math.BigDecimal getPrice();
    void setPrice(java.math.BigDecimal value);
    java.lang.String getServiceURI();
    void setServiceURI(java.lang.String value);
    test.service.OrganizationType getOrganization();
    void setOrganization(test.service.OrganizationType value);

    java.util.List getMethod();
}
```

Attributes are accessed using `getPrice()` and `setPrice(BigDecimal)` (e.g., for price). Class `OrganizationType` represents complex child element `Organization`, and instance of this class is fetched using `getOrganization()` methods. If new `Organization` element needs to be created, `ObjectFactory` is used:

```
ObjectFactory factory = new ObjectFactory();
OrganizationType organization = factory.createOrganizationType();
\\setting specific OgranizationType fields
```

For complex child element with cardinality greater than one (`Method`), another approach is used. The accessor method returns a reference to the live list, and not a snapshot. Any modification made to the returned list will be present inside JAXB object.

Binding compilation and content class generation is done once, at development phase, and after that only when (if) the CDL schema changes. At runtime, process of unmarshalling takes service contract as input and produces set of instantiated Java content classes populated with data parsed from XML document. During unmarshalling contract is validated with respect to schema. The following code instantiates content classes using XML document (contract) from file specified by `path` argument:

```
JAXBContext jc = JAXBContext.newInstance("test.service");
Unmarshaller u = jc.createUnmarshaller();
Contract co = (Contract) u.unmarshal (new FileInputStream(path));
```

After this step, in-memory representation of contract is created. The middle layer uses JAXB to publish and modify service contracts. When a contract is published, Java content classes are persisted in database tables. When a contract needs to be changed, tables are updated, and depending on the origin of update, XML representation is synchronized (via JAXB marshalling).

Web Service Communication

JAX-RPC is used for communication with Web Services. Since middle layer is also realized as a set of Web Services, clients use JAX-RPC to invoke basic functions of the system, and middle layer uses JAX-RPC to invoke single or composite services. In JAX-RPC a remote procedure call is represented by an XML-based protocol (SOAP). Complex SOAP messages and their structure (envelope, encoding rules, conventions for RP calls and responses) are hidden by JAX-RPC API. This API supports development of server side (Web Service implementation) and client side (Web Service invocation) infrastructure. On the server side, remote procedures (Web methods) are specified by writing Java interface and one or more classes that implement that interface. On the client side, a proxy object is created that represents Web Service. All Web methods are invoked on a proxy. Therefore, it is not necessary to generate or parse SOAP messages. The JAX-RPC runtime converts API calls and responses to and from SOAP messages. However, JAX-RPC is not restrictive: JAX-RPC client can access a Web Service not running on the Java platform, and JAX-RPC Web Service can be accessed from non-Java client.

The basic steps for creating a Web Service using JAX-RPC are:

- Coding the service endpoint interface and implementation class
- Building, generating and packaging the required files
- Deploying the WebARchive (WAR) file containing the service

A service endpoint interface declares the methods that a remote client may invoke on a service:

```
package test.service;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ServiceIF extends Remote {
    public int publishService(String path) throws RemoteException;
    public ArrayList allServices() throws RemoteException;
    public int deleteService(int serviceID) throws RemoteException;
}
```

This interface specifies that service will offer three methods: publishing a service, deleting a service and retrieving a list of all services. In addition to the interface, a class must be created that implements it:

```
package test.service;

public class ServiceImpl implements ServiceIF {
```

```
    public int deleteService(int serviceID) {
      //code that deletes a service from directory
    }
    public ArrayList allServices() {
      //code that returns all services from directory
    }
    public int publishService(String path) {
      //code that publishes a service to directory
    }
}
```

After both interface and implementation have been coded, service has to be built. Building a service involves four activities:

- *Compiling a service.* Service interface and implementation are compiled.

- *Creating a service model.* The `wscompile` tool is run which generates `model.gz` file. This file contains internal data structures describing the service. The tool reads the `config-interface.xml` file:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
    <service
        name="Service"
        targetNamespace="urn:Trt"
        typeNamespace="urn:Trt"
        packageName="test.service">
        <interface name="test.service.ServiceIF"/>
    </service>
  </configuration>
  ```

  This file specifies service name (`Service`), its namespace (`urn:Trt`), package containing implementation classes (`test.service`) and service endpoint interface (`ServiceIF`).

- *Packaging a service.* A non-deployable portable WAR file is created (`service-portable.war`) by adding `jaxrpc-ri.xml` and `web.xml` configuration files to compiled classes and service model. The `web.xml` is usual deployment descriptor, while `jaxrpc-ri.xml` looks like this:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <webServices
  ```

```
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0"
    targetNamespaceBase="urn:Trt"
    typeNamespaceBase="urn:Trt"
    urlPatternBase="/ws">
    <endpoint
        name="Service"
        displayName="Test service"
        description="Test service"
        interface="test.service.ServiceIF"
        model="/WEB-INF/model.gz"
        implementation="test.service.ServiceImpl"/>
    <endpointMapping
        endpointName="Service"
        urlPattern="/serv"/>
</webServices>
```

The file specifies service interface, implementation and model, as well as endpoint mapping (name and url patterns) that will be used for service invocation.

- *Deploying a service.* The `wsdeploy` tool is run, and based on the portable WAR creates a deployable WAR by generating required tie classes, generating WSDL file and packaging the tie classes, WSDL file and portable WAR file into single deployable WAR file. After copying this WAR file in the required directory of the application server hosting the Web Service, and assuming that application server name is `appServer`, port is `1234`, context path (required for generating portable WAR) is `service` and url pattern (defined in `jaxrpc-ri.xml`) is `/serv`, the service can now be invoked as `appServ:1234/service/serv/` by interested client parties and its WSDL file is available at `appServ:1234/service/serv?WSDL`.

The described method is used to create a new Web Service and to deploy a server-side part of the solution. Invoking a deployed Web Service from the client side can be done in three basic ways: using static stub, dynamic proxy and dynamic invocation interface (DII). Our solution uses the last option. The reason is that in DII solution a client can call a remote procedure even if the signature of the remote procedure or even the name of the service are unknown until runtime. Those information are obtained during discovery phase and read from directory and/or WSDL file.

Let us assume that following information have been discovered by querying a directory: service name (`Service`), service port (`ServiceIF`), namespace (`urn:Trt`), endpoint (`appServ:1234/service/serv/`), and operation

name (`publishService`). We assume also that operation signature has been discovered (`int publishService(String path)`). In order to invoke an operation to publish a new service, a service factory is created first:

```
ServiceFactory factory = ServiceFactory.newInstance();
```

This object is used to create a `Service` object by invoking the method `createService`. The input parameter of this method is service name:

```
Service service = factory.createService(new QName(qnameService));
```

From the `Service` object, a `Call` object is created. This is actual proxy upon which invocation will be performed on the client-side. A `Call` object supports dynamic invocation of the remote service procedures. This object is created by invoking `createCall` method of the `Service` object with `QName` object representing service endpoint interface as input parameter:

```
QName port = new QName(qnamePort);
Call call = service.createCall(port);
```

Several properties of the `Call` object are set, namely, service endpoint address, URI and SOAP encoding:

```
call.setTargetEndpointAddress(endpoint);
call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
```

Next, input and return types are defined, as well as operation name:

```
QName QNAME_TYPE_STRING = new QName(NS_XSD,"string");
QName QNAME_TYPE_INT = new QName(NS_XSD,"int");
call.setOperationName(new QName(BODY_NAMESPACE_VALUE,"publish"));
```

Input parameter is passed to the method and the operation is executed with input parameter `path`:

```
call.addParameter("String_1",QNAME_TYPE_STRING,ParameterMode.IN);
Integer result = (Integer) call.invoke(new Object[] {path});
```

Asynchronous (one-way) calls are implemented by using `invokeOneWay` instead of `invoke` method:

```
call.invokeOneWay(new Object[] {path});
```

Composition and Correctness Verification

The functioning of JAXB and JAX-RPC runtime is shown in Figure 7.7. It shows two typical use cases: publishing a new service to directory and composition of two services that are already in directory. Prior to any client calls, XSD schema describing Contract Definition Language is compiled with JAXB binding compiler, and content classes are stored in the middle layer. Client publishes new service by issuing SOAP or JAX-RPC call to the Publish Proxy, which delegates the call to the Publish service in the middle layer. A service that is to be published is located, and its CDL description is unmarshalled into precompiled content classes produced by JAXB compiler. Finally, write to underlying database is performed via JDBC which completes the publish process.

Composition is initiated by sending SOAP/JAX-RPC request to Compose Proxy, and the call is then delegated to Compose service in the middle layer via JAX-RPC. It processes composition request, retrieves partner service information from database using JDBC, verifies composition correctness by calculating function *correct*, and constructs required dynamic proxies that represent partner services using Dynamic Invocation Interface. In case of automatic composition, the chosen algorithm for automatic composition is run on the target contract and potential composition partners are identified and passed as parameters to the Compose Proxy (this case is not shown in Figure 7.7). Each individual proxy then connects to its implementation and middle layer coordinates message passing in a manner that depends on the composition pattern used (Figure 7.8). For example, if sequential composition is required, middle layer will invoke one service in the manner described in the previous section, retrieve its results and pass them to the next service in chain; if parallel composition is required, each service will be invoked independently, with synchronization logic that will compare/select result. Result is then returned to the client via Compose Proxy.

All instances of parallel composition (parallel with and without communication, choice) require separate logic for coordination, which is not trivial. Denoted with *fork* and *join* in Figure 7.8, this logic is implemented using native Java multithreading and synchronization. This additionally burdens composition server, especially when many compositions are processed at the same time: many threads will compete for the resources and synchronization becomes difficult. A more natural solution would be to use asynchronous invocation of target Web Services, effectively decoupling composition server from the parallel services. However, the problem is notification mechanism which must be separately provided. One solution is to use polling, where composition server assigns a unique identifier to every request sent to asyn-

chronous services, and then periodically polls them using that identifier. The downside is that polling infrastructure must be provided (management scheme for identifiers and results) as well as the fact that polling is not very efficient. The other solution is to use a complete message-oriented system such as Java Message Service (JMS) where composition server would put a message in a request queue and would subsequently receive message with results in a response queue. Upon receiving all response messages, synchronization and/or subsequent composition are performed. At this point it was determined that second and third approach (polling and JMS) are too complex for the current composition server implementation. Therefore, native multithreading was selected for performing parallel composition, having in mind that downsides of the selected method can be ideally aleviated by using asynchronous invocation with message queuing system.

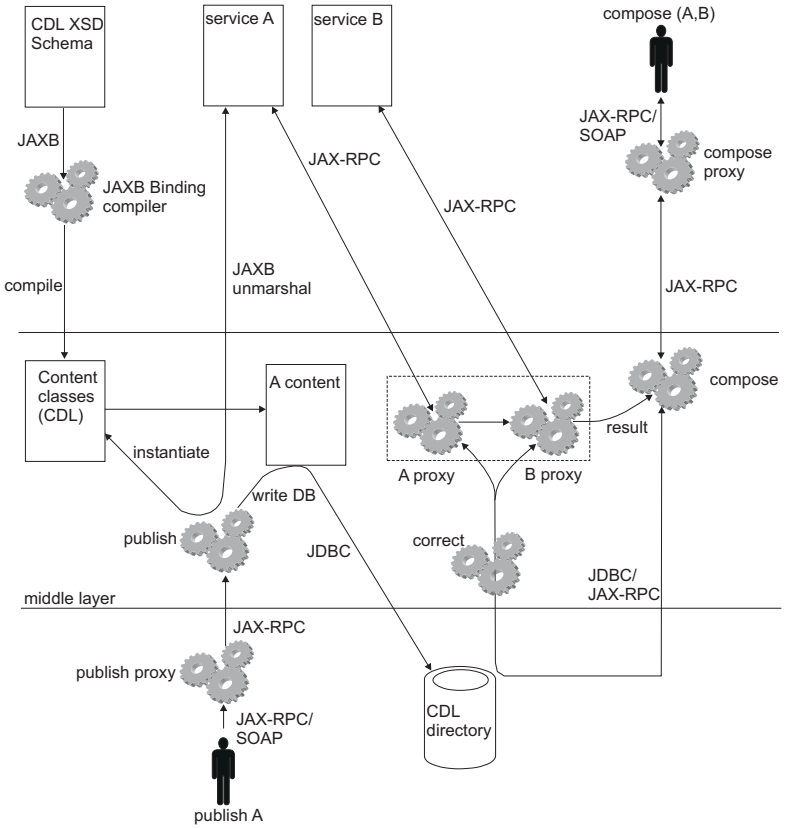

Figure 7.7: Possible Scenario of JWSDP Runtime

Figure 7.8 shows implementation of all composition operators. Operation *map* performs mapping of input variables to service ports, thus making pos-

sible various parameter combinations and aggregations. Difference between parallel and choice composition is now evident. Parallel composition without communication performs forking into separate threads that will make their results available as soon as each service finishes execution and will never synchronize. The subsequent services can use the results as soon as they appear. On the other hand, parallel composition with communication will also fork into separate threads, but will be synchronized and logical operation on their outputs will be performed, effectively selecting one output and discarding the other. Finally, choice composition will execute both services in parallel, but without message mapping, since choice requires that both services should be compatible and accept the same input parameters. After synchronization (join) is performed, both results are available to the next service. Instead of introducing asynchronous choice operation, parallel composition without communication can be used with both services being passed identical set of parameters. Naturally, subsequent synchronization of such composition can be performed using parallel composition with communication.

After Compose Proxy retrieves partner services' description from the directory (or automatic composition algorithm identifies partners), verification of correctness takes place before execution. In case that verification fails, actual invocation is not performed. The module for verification itself was not developed, since there are many existing solutions already offering this functionality (e.g., B-Toolkit, Atelier-B or B4Free). However, the chosen module was ProB [87], an open-source model checker for the B-method. It is Prolog-based proving engine, with Java interface, which was the main reason for choosing it, as easy integration into the remainder of the system was important. It uses co-routining and finite domain constraint solving to make animation of B machines possible [88].

As Figure 7.7 shows, the composition process is abstract from the user's perspective. It is performed on the abstract machines and service proxies. Only after the composition has been verified and proxies have been arranged, will the middle layer actually invoke and execute target service implementations. This solution brings additional benefits, as it is dynamic and fault-tolerant, enables easier replacement and upgrade of service implementations and offers a higher level of decoupling between user, service description and service implementation. Composition is therefore independent of concrete service implementation and binding to implementation (service instance) will be performed late in the composition lifecycle.

Figures 7.5 and 7.7 also show that middle layer is completely accessible via SOAP / JAX-RPC, which means that any Web Service client can use functionalities that it provides. It furthermore means that entire composition server can be recursively used (re-composed) in a complex service-oriented

Composition operator	Runtime implementation
$A \triangleright B$	$tmp = invoke(A(map(input))$ $res = invoke(B(map(tmp))$
$A \| \| B$	$result_A = fork(A(map(input)))$ $result_B = fork(B(map(input)))$
$A \|\|_P B$	$fork(tmp_A = A(map(input)))$ $fork(tmp_B = B(map(input)))$ $join(A, B)$ $if(P) \;\; res = tmp_A \;\; else$ $res = tmp_B$
$A \odot_C B$	$if(C) \;\; res = invoke(A(map(input))$ $else \;\; res = invoke(B(map(input))$
$A \square B$	$fork(tmp_A = A(input))$ $fork(tmp_B = B(input))$ $join(A, B)$ $res_1 = tmp_A$ $res_2 = tmp_B$
$\circlearrowleft_{P(e)} A(e)$	$while(P(e))$ $res, e = A(map(input))$
$W(e) \circlearrowleft_{P(e)} A$	$e = W(map(input))$ $while(P(e))$ $res = A(map(input))$ $e = W(map(input))$

Figure 7.8: Implementation of Composition Operators

application, assuming that a directory stores description of all operations that composition server offers.

7.3.3 Directory and Searching

Service directory is realized as MySQL database, and is addressed by the middle layer via JDBC. Therefore, any other relational database can be used instead. There are several reasons why a relational database is used instead of a native XML database. Current XML databases still do not support W3C XML schema which is used to define CDL. Using native XML database could therefore lead to low data integrity. Furthermore, XML databases use XPath as query language, and it offers no support for grouping, sorting, cross document joins, and data types. Since service directory requires complex queries, this is a very limiting implementation factor. Still another downside

is that updating requires retrieving an XML document, modifying it using own API and then returning it to database.

Database was designed to take full advantage of rich descriptive options offered by CDL in order to overcome UDDI limitations. The basic entity in UDDI is not a service, but an organization. Services belong to organizations. UDDI offers searching organizations by name, descriptions (essentially keywords) and classification. Only after adequate organization has been found, services belonging to it can also be searched. There is no way to search for services directly. It is also possible to search for services belonging to organizations by the names of their WSDL documents. This all means that using UDDI it is impossible to search for services offering certain operations, accepting certain parameter types or having some other property [1].

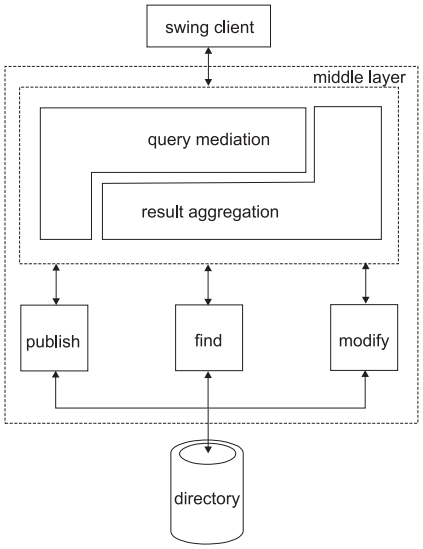

Figure 7.9: Accessing and Searching a Directory

The underlying database schema still retains notion of organizations and services belonging to organizations, but allows for searching services directly, using any combination of properties defined in CDL (database schema is given in Appendix F). That means that it is possible to search for services by locations, methods they offer, classifications, and all other properties defined in their pre-conditions, post-conditions and invariants. One example query

[1] It should be noted that UDDI v3 enables distributed directories and mapping of semantic descriptions to UDDI structure using `uddi:categoryBag` element, although still with not much convenience, since it requires additional (customized) requests for extraction of the whole ontology [123]

would be to find all services in the 200 m radius that accept postscript documents and print them in color with 1200 dpi resolution, free of charge if a client can supply security credential of a certain type. Besides being a clear advantage compared to UDDI when searching single services, the ability to perform such complex queries is very important when searching adequate composition partners.

Figure 7.9 shows architectural solution for directory interaction. Three types of queries are mediated in the query mediation: publishing new services (`INSERT`), finding existing services (`SELECT`) and modifying existing services (`UPDATE`). Query mediator contains a set of classes that persist JAXB object (content classes representing instantiated contract) into database tables. Those classes then use publish interface to perform actual write into database. When performing a search, query mediator constructs adequate query using finder classes. Each finder returns its own result to result aggregation, which then performs necessary cross joins. End search result is returned to the client. Finally, modification is performed using finders to locate all points (tables) for updates, and then actual modification is performed through modification classes.

7.3.4 Transaction, Exception and State Management

In order to ensure correct execution of composed Web Services, several infrastructural requirements must be satisfied, namely distributed transaction, exception and state management.

7.3.5 Transaction Management

Implementing transactional behavior is essential for correct functioning of composition framework. Web Services run on heterogeneous platforms that have different characteristics such as transactional support, concurrency policies or access rights. Web Services can use different transport protocols which involves inherent media unreliability. Web Services can be unavailable for an unknown reason and for an unknown amount of time. Due to these facts, composition framework requires dedicated fault tolerance mechanism [161]. The isolation of services raises challenges in treating behavior of composition in the presence of faults. Many times it will be required to restart or cancel execution of particular composition due to a fault. The possible faults can be:

- Internal service error (level of constituent services)
- Fault of the underlying platform (hardware, application server)

- Communication failure (network and timeout)
- Composition server error or inconsistency
- Online upgrade of constituent services and their application servers

Transactional support ensures that execution is monitored and can be safely aborted and/or re-executed at any point. Transactions are used to return a system to a previous stable state in which it was before an error occurred. It is therefore a backward recovery mechanism. The basic structuring units for providing fault tolerance of distributed systems like Web Services are distributed transactions[58]. They have been proven very successful when applied to closed distributed systems, and as such can be used to enforce consistency of single Web Services (inside their application containers).

However, distributed transactions are not entirely suitable for enforcing ACID (atomicity, consistency, isolation, durability) properties upon compositions of services for two main reasons. First, transaction management over composition of Web Services requires cooperation among transaction support frameworks of each particular service. Information required for such cooperation is not contained in WSDL document. Second, locking resources until the entire transaction commits is not well suited for service architecture, since other clients will have to wait until a resource becomes free. The problem is that Web Service composition requires transactions that can be of variable durability, some being measured in days.

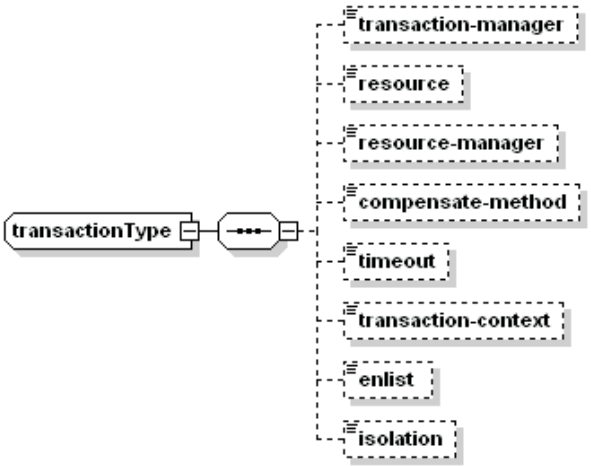

Figure 7.10: CDL Transaction Specification

In order to address the first problem, CDL introduces `transaction` element (Figure 7.10). It describes underlying transaction mechanism of a single Web Service. Properties of a transaction mechanism are transaction manager (application server hosting Web Service), resource (persistent storage such as but not limited to relational databases), resource manager (driver that is used to connect to persistent storage), transaction timeout, enlist modes (service can support, require or join transactions) and isolation level (concurrency control: read uncommitted, read committed, repetable read or serializable).

Since this information can be specified in pre-conditions, post-conditions and invariants, it is used in several semantic contexts. If a transaction element appears in service pre-condition, it expresses service's transactional requirements, that is, minimal conditions under which it will execute in a joint transaction with partner services. If a transaction element appears in post-condition, it expresses transaction capabilities that will be provided if a service is to execute inside a joint transaction. Finally, if a transaction element appears in invariant, it specifies properties that must not be violated by transactional behavior of other services.

However, different transactional backgrounds may not be compatible with each other, or even when they are, some services may not be willing to comply with transaction request given their own transaction policy (e.g., enlist mode set to never join a transaction or always require new transaction). In order to address this issue, as well as to improve the problem with long-term resource locking, a second transaction mechanism is introduced, namely split (open nested) transaction model[107], shown in Figure 7.11.

In order to eliminate resource locking for unnecessary long periods of time, one (potentially long) transaction can be split into a number of concurrent "smaller" subtransactions that usually correspond to single Web Service methods, but can also be another compositions. Typically a subtransaction matches a transaction supported by the underlying framework in which each Web Service is executing. Each subtransaction can commit independently of others, in any temporal order. That way, resources are freed the moment each service finishes its operation. In order to enforce atomic operation execution though, compensating (undo) actions are required for every subtransaction. If one subtransaction aborts, others that have already committed must compensate their actions. Since the final write to persistent resource has already been done, a separate operation must be provided that will compensate for this action and restore consistency. This operation is called compensating operation and must be provided with adequate context to execute correctly.

Context maintenance in the case of split transactions is not a trivial problem, especially if other persistent storages apart from relational databases are

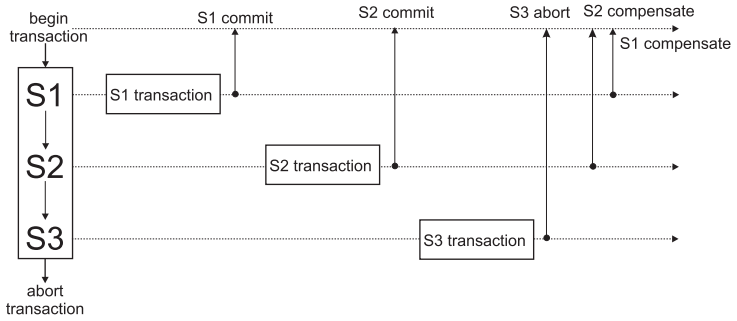

Figure 7.11: Split (Open-nested) Transactions Model

allowed. Therefore, context maintenance protocol is introduced. All transactions performing concurrent operation upon the same persistent resource are enclosed in a hierarchical scope. Until transaction on top of the scope commits, context of all other transactions in the same scope is kept. In case any transaction in the scope aborts, the protocol performs the following actions:

- Aborted transaction is removed from the scope.
- All transactions that have already committed are left in the scope, but without compensation and/or restart.
- All transactions that have not yet committed are aborted, compensated and restarted.
- A transaction is removed from the scope once it commits and there are no other transactions above it in the same scope.

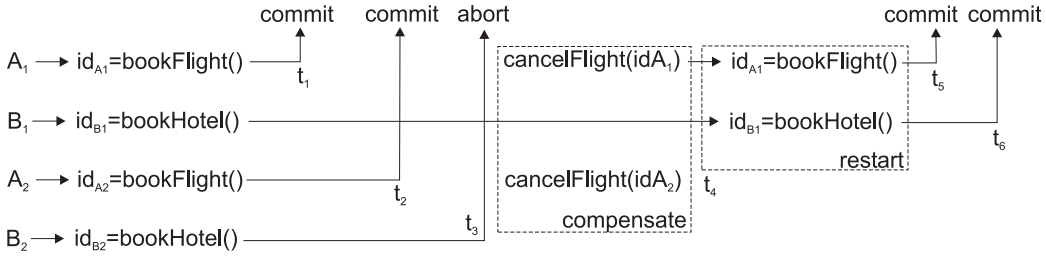

Figure 7.12: Split Transaction Protocol

Figure 7.12 shows an example of the protocol application. Two services, `A` and `B` are composed in parallel by two different clients. Services offer operations `bookFlight()` and `bookHotel()` respectively. The first client invokes

both services, and in t_1 method `bookFlight` for the first client commits. Transaction context, which is the id of the booked flight $\texttt{id}_{A1}$, is stored. In t_2 method `bookFlight()` of the second client also commits, and its context is stored in $\texttt{id}_{A2}$. In t_3 method `bookHotel()` of the second client aborts for some reason, while method `bookHotel()` of the first client is still running. Protocol now enters compensation phase. Compensation method for the second (aborted) client is invoked. This is possible because its context $\texttt{id}_{A2}$ was stored. After that the second client's transaction is removed from the scope. Since the first client's transaction did not commit entirely, it is also compensated by aborting uncommitted methods and compensating committed `bookFlight()` method by invoking `cancelFlight(`$\texttt{id}_{A1}$`)`. Once compensation has been performed, transaction is restarted in t_4. After both methods of the first client commit in t_5 and t_6, transaction is removed from the scope.

The way to associate Web method with particular persistent resource is part of the state management mechanism, described in Section 7.3.7.

7.3.6 Exception Handling

Apart from transactions, which belong to backward error recovery techniques, a forward error recovery mechanism is also necessary for Web Service composition. The difference between backward and forward error recovery is that the former returns a system to the previous stable state, while the latter moves a system to any stable and thus correct state [85]. The reason why Web Service composition framework requires forward error recovery is the fact that given the heterogeneity of their applications, actions of Web Services are sometimes irreversible: operating on external devices, movement of goods, operation upon environment, or real-time systems.

In modern software systems, forward error recovery is usually implemented using a suitable variation of exception handling mechanism [36]. Similar approach has already been implemented for Web Services, namely in BPEL4WS which allows for definition of exception handlers (called fault handlers) that can be associated to an activity (scope). When an error occurs inside a scope, execution is terminated and corresponding fault handler is invoked. Scopes can be nested within each other. When a scope is running concurrently, and some process inside signals an exception, all other concurrent processes are terminated, but only one handler is executed: that for a process that signaled an exception. Therefore, such error recovery treats only signaling process and cannot guarantee that other concurrent processes (services) will continue executing in a correct context, unless a single exception handler can account for all operations involved.

In order to overcome these difficulties, coordinated exception handling is

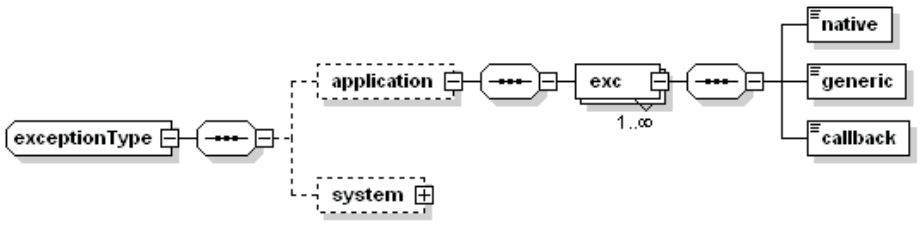

Figure 7.13: CDL Exception Specification

introduced [162]. In coordinated exception handling, all participants of the composition are involved in dynamic and cooperative handling. If several exceptions have been raised concurrently, they are resolved hierarchically. Exception scopes are also defined dynamically, based on declarative exception handling properties stated in service contracts (Figure 7.13).

Two types of exceptions are defined: application and system level exceptions. Application level exceptions include exceptions cast by Web Services themselves, signaling internal service failure (programmed exceptions), while system level exceptions are cast by the composition server, signaling system faults like database crashes or network failures. For each exception its native name, generic (wrapper) name and callback function are specified. Generic name is derived from common namespace (example in Figure 7.14) and it is used in cooperative handling to determine whether a certain service is able to handle specific exception. If it is, callback function is invoked which subsequently handles the exception.

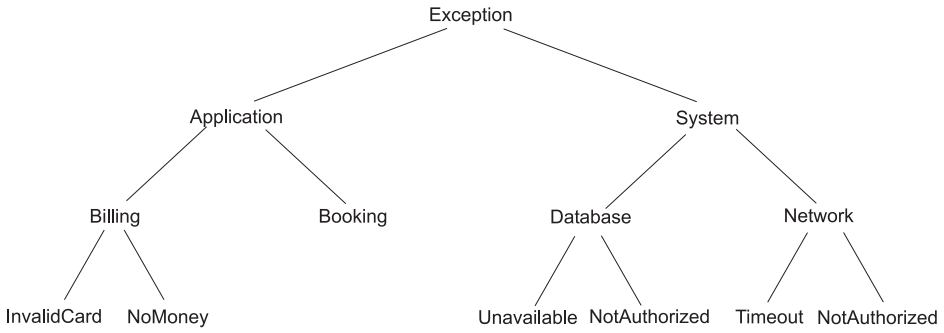

Figure 7.14: Common Exception Hierarchy

Exception blocks (scopes, equivalent to `try...catch` blocks) are built dynamically, using conservative hierarchical nesting. Figure 7.15 shows exception block nesting for a travel reservation scenario from the previous sec-

tion.

Exception handling for this composition is implicitly managed like this:

```
try {
  user;
  try {
    flight || hotel;
    try {
      pay;
    } catch (pay)
  } catch (flight || hotel)
} catch (user)
```

However, contrary to the classical exception handling protocol, where exception is forwarded to the next level only if it was not appropriately handled at the current level, in cooperated exception handling all parties receive notification of the exception regardless of the level their handler is situated at. Cooperative handling is performed by invoking appropriate callback functions (if they exist) and at the end of the handling process a group decision is taken whether the handling was performed successfully or not. Several application dependant strategies can be used to judge the handling process:

- All involved participants have to handle the exception
- Any participant has to handle the exception
- At least exception source has to handle the exception
- At least one service in every scope has to handle the exception
- At least initiating service (client) has to handle the exception

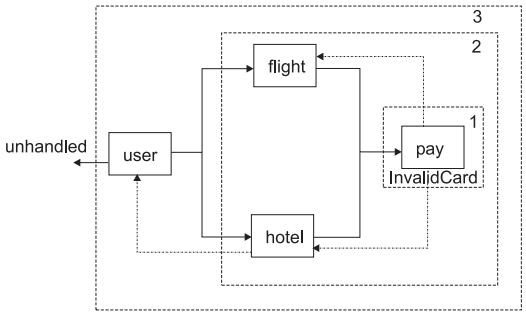

Figure 7.15: Cooperative Exception Handling

Figure 7.15 shows a scenario where exception source (`pay`) is unable to resolve the `InvalidCard` exception, and forwards it to services `flight` and `hotel` in the parent scope. Service `flight` handles the exception successfully while `hotel` is not able to do so and forwards it to its parent scope to `user` which is also not able to handle the exception. All participants are now informed about the exception and based on the chosen behavior exception is either pronounced as handled and execution is resumed, or it is sent to the runtime environment as unhandeled and execution is suspended.

7.3.7 State Management

Up to now we have been talking about modeling Web Services using abstract machines comprising state variables. It is obvious that it was implicitly assumed that some services can maintain their state between calls. However, Web Services are stateless and state management mechanism needs to be introduced.

Although Web Services are inherently stateless, many of them allow for the manipulation of the state, such as persisting data into databases, file systems, or coordinating dependent messages. There is ongoing debate in the community whether Web Services should or should not support state management. One view is that Web Services are not another Object Request Broker architecture, and therefore should have no notion of state [171], while the other view is that state management plays the critical role in distributed computing and as such must be addressed at the architectural level [43]. The former point may be true at the fundamental level of Web Services (discovery, description, invocation), but our position is that for the purpose of complex service interactions the latter view is correct.

Why is state management important? Let us observe a holiday booking scenario implemented using Web Services. One part of the holiday planing is flight reservation. Suppose there is a Web Service offering following Web methods: `bookFlight`, `modifyReservation` and `deleteReservation`. Although all methods are stateless, they all interact with stateful resource in the background, e.g., with a relational database. The method `bookFlight` will create a record in the database and return booking id, `modifyReservation` will modify a record based on the id passed to it, while `deleteReservation` will delete a record. State is implicit here, since all methods change the underlying persistent resource. There are two possible ways to associate a state with a Web Service [175]:

- A conversational service implements a series of operations where result of one operation depends on the prior operations of the same or other

services. The state is maintained in the logical sequence of messages.

- A service that acts upon one or more persistent resources (database, file), creating, modifying or deleting it based on the messages it sends or receives.

Since conversational state can be implemented using WS-Coordination and WS-Context specifications, we concentrate on the interaction with stateful resources. Furthermore, we consider only relational database as a provider of background persistent resource. Interaction with persistent resource is described within the `resource` element of the CDL (Figure 7.16).

Resource is identified by its name, uri, and resource manager (in our case, relational database driver). For each method acting upon a resource, one of the following actions can be defined: `CREATE`, `READ`, `MODIFY`, `DELETE`. Methods that create resources return resource identifier, while methods that read, modify and delete resources require resource identifier. Finally, resource property defines one or more CDL elements (state variables) that are bound to the underlying resource.

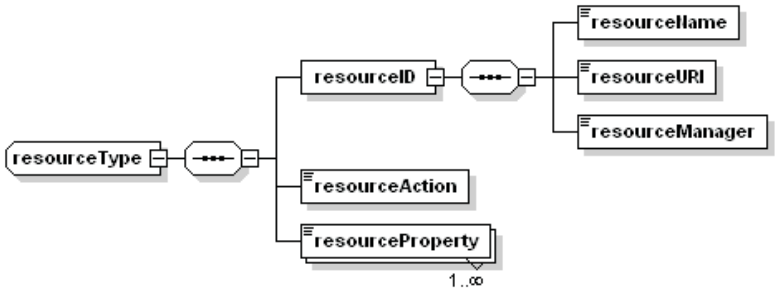

Figure 7.16: Description of Underlying Persistent Resource

Our efforts in providing state management are compatible with the recent WS-Resource proposal [44], with the main difference being that WS-Resource supports broader range of persistent resources identified using WS-Addressing. Since we currently support only databases, resource identifier comprising name and uri (actually server and database name) is enough. However, it will be possible to incorporate WS-Resource elements into CDL once this specification is standardized.

7.3.8 Composition in the Presence of Failures

Once the execution of a composite service has started, several things can happen, e.g., a component service becomes unavailable, or parameters (con-

tract) of a component change because of the on-line update. The situation is further exacerbated by the fact that the nature of Web Service interactions is such that sometimes it takes days for a composition to complete. Therefore, the probability that one or more similar problems will occur cannot be neglected. It is necessary to enable adaptation to changes that occur during the execution of a composite service, by revisiting the execution plan to revalidate composition and/or replace some components. This is necessary in order to ensure that promised quality of service remains optimal, or that composition is possible at all.

Assume that a composition started at time t_0 and the upper bound of its execution is $t_0 + t_{WCET}$. For the purpose of handling composition in the presence of service failures, for a given time t, participating services are partitioned in those that have completed in time interval (t_0, t), those that are being executed in t, and those that are scheduled for execution in $(t, t_0 + t_{WCET})$, but have not yet been executed. These partitions are called S_{FIN}, S_{EXE} and S_{SCH} respectively. The execution of a composite service may be replanned in two cases:

- Exceptions have occurred in the section S_{EXE}
- Changes have happened in the section S_{SCH} causing components either not to be able to deliver their specified properties (service contract has changed) or to become unavailable.

The second case is handled automatically by the composition server. During execution, services in the section S_{SCH} are constantly monitored, and as soon as one or more of them issue a write to the directory, execution is suspended. Update to database can either mean a new contract has been published, or that service has become temporarily unavailable due to on-line update (this is detected using a separate field in the database). In case service contract has been changed, composition is revalidated, taking into account that services from S_{FIN} have already executed and cannot be replaced. If a new contract violates correctness and/or deviates from original composition plan, a query to find a substitute service is issued to the directory. The same happens if service becomes unavailable. If no substitute service can be found, composition is rolled back and aborted.

In order to be able to deal with the first case, where services fail during the execution and need to be replaced, an additional method is added to all exception handlers: `replace(service)` that suspends composition and tries to find a substitute service for the one that has cast an exception:

```
try {
```

```
  serviceMethod();
} catch (TimeoutException e) {
  replace(service);
} catch (UnavailableException e) {
  replace(service);
} catch (Exception e) {
  //perform custom (user defined) handling
}
```

In case a service has timeouted (`TimeoutException`) or has become unavailable by setting adequate field in a directory (`UnavailableException`), another service will be provided for to replace the failed one. In other cases of failures, custom exception handling is performed.

7.4 Peer to Peer Extensions

The presented solution for Web Service composition is centralized, as it requires a composition server to manage execution of partner services. This is the case with most other service composition approaches, with the notable exception of [20, 152]. On the other hand, participating Web Services are inherently distributed and autonomous. The centralized execution monitoring model can potentially introduce scalability, availability and security problems [31]. Taking into account the above facts, it is reasonable to investigate whether a proposed model can be implemented without requiring centralized server, that is, whether it is possible to modify implementation so that Web Services coordinate composite process execution themselves, in a peer-to-peer fashion.

The following issues can be solved in a peer-to-peer environment:

- Verification of correctness
- Message exchange
- Exception handling

Correctness verification can be implemented by creating a separate Web Service (verificator) that receives abstract machines (service contracts) as input and calculates the value of the function *correct*. This service is composed sequentially to the entire composition at its beginning. If the verificator returns false, entire composition is aborted, otherwise it is executed. Naturally, since many compositions will require simultaneous calculation of correctness,

a pool of verificator instances can be created in several independent application servers and load balancing can be used to guarantee the appropriate response that will not significantly impact execution of the composition itself.

Peer-to-peer message exchange is the crucial element to solve. In the centralized solution, all messages are intercepted and interpreted by the composition server, that decides which service(s) will be executed next, in what order and with what parameters. Composition server thus acts as a dedicated service scheduler that implements scheduling according to the composition patterns being used. In a peer-to-peer execution model responsibility of managing message exchange among participating services lies with the services themselves. Obviously, this is not possible with the current way Web Service architecture manages service interaction, but peer-to-peer extensions can be proposed. Each service should have a component associated to it that manages message coordination. Let us call this component the coordinator and examine its role and capability. The role of the coordinator is to initiate, control and monitor execution of its associated service. In order to do this, it has to have an insight into service contract and be able to communicate with coordinators in charge of other services for a given composition. The main task of the set of coordinators is to implement distributed scheduling policy based on the behavior of participating services, in other words, to determine when a certain service is activated (executed), what are its inputs and what should be done with its outputs. The main activities of a coordinator are to receive notification of execution completion from other coordinators, to use these notifications to determine when its own service should be executed, and to notify other coordinators when its own service completes. A composite execution is performed by peer-to-peer message exchange between coordinators. In effect, distributed scheduler can be implemented using deadline-constrained causal ordering protocols [145] that ensure that if two messages are causally related (as determined by composition operators used), they are delivered to the receiver in the logically correct order. Causal ordering protocol prevents causal order violation thus enforcing composition integrity. This requires that all messages have defined deadlines. Additionally, causal ordering of messages can be implemented using distributed shared memory scenario [4, 133] (similar to centralized blackboard approach), where it is important to optimally delay writes in order to achieve logical order [13].

Finally, exception handling can also be performed by coordinators, by sending notification of events other than successful completion (e.g., exception, unavailability, external communication error, data access problem) to other coordinators. Also, based on the received notification, the coordinator should decide whether to execute, cancel, abort or compensate its service.

The following issues are, however, very difficult to implement in the pure

peer-to-peer environment:

- Transaction management
- State management
- Directory (metadata)

Open nested transactions are difficult to implement even when there is a central orchestrator that maintains log, performs checkpointing and manages execution context in order to enforce correct undo operations (compensations). Even more problematic is enforcing ACID properties based only on services' native transactional capabilities, which is sometimes not possible at all due to the limitations of each service and/or its home application container. Transactional logic could be implemented inside coordinators, or separate component that is associated to each service. However, while coordinator can implement relatively simple scheduler, building and executing peer-to-peer transactional support component would involve too big overhead. Optionally, those components could be built without support for the open nested transaction model, but that would introduce two new problems: first, the benefits of open nested transactions are lost (resource locking), and second, a consensus protocol would have to be implemented instead, because all services would have to vote at the end of the transaction in order for it to be committed. In a volatile environment in which Web Services are executing, developing such consensus protocol presents a major difficulty [16]. However, there are approaches to achieve consistency in peer-to-peer systems, namely [14] presents a quorum-based system for implementing peer-to-peer mutual exclusion. This solution may be used to enforce ACID properties.

State management is not critical issue because it is mainly used to support open nested transaction model. In case that open nested transactions are not used, state management is not that important. Also, it can be implemented by adding additional capabilities to coordinator component, namely the ability to remember database tables and primary keys that were used to access persistent storage and to share them with other coordinators when necessary. This information can then be used for scheduling.

Perhaps the most difficult problem to solve in a peer-to-peer environment is that of a true distributed and shared directory. In a peer-to-peer network data is stored on the nodes and replicated to insure integrity. In our model, each service, apart from its own contract, can store contracts of other peers, depending on its storage capacity. Data lookup in a peer-to-peer network is usually performed using distributed hash table (DHT) [12]. For every document stored in a network, DHT value is calculated which uniquely identifies

the document. When the value of the hash function is known, the query is performed by simple routing through peers that have the required data, as they are directly identified by the hash value. However, this works only in the case that entire document is searched for, meaning that searching for the partial information is very difficult, not to mention complex queries like joins. Currently, even locating a file in a file system has not been successfully solved in peer-to-peer systems. Therefore, creating a peer-to-peer service directory presents a major challenge that might be very difficult to solve in the near future. This is, together with transactional support, perhaps the most limiting factor in porting the presented solution in true peer-to-peer environment.

Chapter 8

Conclusions

8.1 Contributions

Taking historical perspective into account, service-oriented architecture is a logical step initiated by the recent developments in the areas of distributed computing and business process modelling, as well as increasing ubiquity of networking technologies. The main goal of SOA is to introduce standard methodologies, languages and protocols for development of distributed applications out of loosely coupled, independent and autonomous software entities.

With the rapid advance of service-oriented paradigm in many areas of information systems, not limited to enterprise application design, development and maintenance, but applied to embedded systems and network architectures as well, plethora of proposals for standards have appeared and somewhat clogged the architectural point of view of the whole paradigm. One of the problems in service-oriented computing that has still not been satisfactorily solved is the way of creating new services and applications out of existing, predefined ones. This process is called service composition. Therefore, the goal of this dissertation was to advance state of the art in service-oriented computing through design and development of composable service architecture. From author's perspective the main contributions of this work are:

- *Specification of services' non-functional properties* with introduction of contract-based framework for service description (pre-conditions, post-conditions and invariants) and isomorphic description notation (XML-based Contract Defintion Language and Abstract Machine Notation).
- *Enhanced directory capabilities*, required for identifying adequate composition partners.

- *Verification of service composition correctness* through introduction of formal composition language (composition operators applied to abstract machines) and verification criteria (type checking, composite invariant preservation, correct termination and feasibility).

- *Web Service Design Patterns* as high-level composition constructs that offer verifiable components for design of service-oriented applications.

- *Automatic service composition* solved by treating it as a search problem with several search strategies proposed (basic heuristic search, probabilistic search, learning, decomposition and bidirectional search).

- *Contract-based composition server design with exemplary implementation*, using Java-based technologies with specially developed support for composition infrastructural requirements, such as distributed transaction, exception and state management.

Figure 8.1 summarizes state of the art in service-oriented computing and additional properties offered by the composable service architecture.

The issue of *evolution* merits additional discussion. Although Web Services promise loose coupling, they are mostly used for remote procedure calls (RPC) in today's service-based applications. This incurs problems when new version of the service is deployed, which is a well-known issue in RPC-based systems. Updates, evolution and versioning of clients and services have to be synchronized and coordinated in such environment, otherwise application will face serious problems. On the other hand, Web Services should decouple clients and services, but since static invocation (using pre-generated stubs) is the preferred method of issuing remote calls [71], this architectural benefit is lost due to implementation-time assumption of a static service description (static WSDL). Once WSDL document changes, new set of stubs has to be generated and client application rewritten, recompiled and redeployed. The core of Web Services design should be centered around *programming without assumptions* paradigm instead: nothing is assumed in advance, everything is discovered on-demand. Using contracts as boundaries in dynamic invocation scenarios allows independent evolution and versioning of clients and services since clients can either adapt if the contract of the target service has changed (renegotiation) or can search for another partner if new contract does not fit client expectations anymore. Verification and automatic composition ensure correctness of such an approach.

	State of the art in service-oriented computing	**Composable service architecture**
QoS	Little or no support for non-functional parameters comprising QoS and SLAs	QoS descriptions are integral parts of service contract
Reuse	Based on WSDL and informal service description	Verifiable compositions and design patterns facilitate service reuse, opening a possibility of service marketplace
Correctness	Judging composition correctness at design time is difficult	With the use of AMN, design and development of composite services facilitated by automatic proving
Semantics	Clients must know atomic service functionalities	With automatic composition there is no need to know properties of all atomic services, only goal properties
Adaptivity	Manual, hard-coded partner selection	Flexible, automatic on-demand selection
Availability	Manual replacement and failover	Offers infrastructure for automated reliability guarantees, as long as there are available services that can be automatically substituted
Evolution	Clients and services have to be developed and deployed mostly simultaneously because of strong dependencies	Independent deployment of clients and services by using contracts instead of types for boundaries

Figure 8.1: Comparison of SOA state of the art and composable service architecture

8.2 Crossing the Infrastructures

"If the film The Graduate were remade today, the word of career advice whispered in Dustin Hoffman's ear might well be *services* instead of *plastics*" [69].

Is this a gross overstatement? For sure, plastics is everywhere around us, it can be physically perceived (touched, smelled) and plastic products follow us throughout our everyday business or leisure routines. How about services? They cannot be seen or felt. They cannot be manually used. They cannot be even broken in rage. However, let us observe the following facts:

- Services now account for more than 75% of the U.S. economy. [1]
- The population and the labor force have shifted dramatically away from farms to cities, from fields to factories, and, above all, to service industries (81% of non-farm employees). [2]

In the investigation of the relevancy of the above claim (plastics and services), one needs to take a broader view with respect to what service-oriented computing is. It represents much more than current technologies (e.g., Web Services) or textbook examples (e.g., travel reservation scenario). *Service industries* today comprise transportation, utilities, wholesale and retail trade, finance, insurance, real estate, government, etc. The new work force is migrating from old industries thus fueling the rapid expansion and growth in these areas. This expansion and globalization have not been adequately followed by the supporting information infrastructure. Hence the need to develop and apply service-oriented principles not only in narrow and limited technology-oriented domains, but also in each and every domain mentioned above.

Elements of the composable service architecture can bind information society components (services) and infrastructure (communication and verification). Exposing embedded systems and existing software utilities as Web services and enabling verification of correctness on a large scale could cause rethinking the way global operations are transacted, but also the way applications are developed and deployed. The process of migration towards service-oriented computing has already started and composability as a property of an architecture can enforce security, dependability and usability.

The need for deployment of service-oriented architectures on the large scale is evident in the globalization of the modern world. In the new economy, characterized by enterprise and business integration on the unprecedented

[1] IEEE Technical Committee on Services Computing

[2] US Department of State report

scale, the ability to allow late and short term binding of business processes opens a possibility of a true service marketplace. Apart from the design-time benefits for composite service application designers, composable service architecture offers additional properties to the service marketplace, such as service-level agreement negotiations (contract-based description), adaptivity (reuse, automatic binding) and trust (verification of correctness, reputation systems).

Let us consider electric power industry and try to investigate if and how composable service architecture can be utilized in this domain. Electric power grid has evolved in the past decades to become one of the most complex systems ever built. Deregulation has caused separation of roles and domains, such that only in Germany, for example, more that 1000 utility organizations cooperate in the national power grid. The demand for higher-quality power is ever rising and generation is being increasingly distributed, too. The supporting grid communication infrastructure however, has not followed these developments. Currently, communication infrastructure for the power grid (used for receiving status and sending control data) is several decades old and based on inadequate star topologies using outdated Supervisory Control And Data Acquisition (SCADA) systems. As such, this infrastructure is not able to reflect the distributed and dynamic behavior of the today's power grid, resulting in the recent blackouts in the North America, Europe and Asia, that affected several hundred millions of people. The main problems with the existing communication infrastructure are:

- Inadequacy: Data acquisition and response are slow and there exists unawareness across company and regional boundaries.
- Inflexibility: The existing infrastructure is rigid, companies involved are inert, with new information technologies and approaches being difficult to introduce in the existing environment.
- Cost: Extra point-to-point communication links, necessary to establish more flexible communication, are very expensive.

The new requirements, on the other hand, are:

- Where once a centralized company was established, many new actors appeared: substations, transmission lines, power generators, energy marketers, customers, smart acquisition devices, underlying network technologies; all of them are owned and administered by different parties.
- Actors are separated by legal, safety and technological barriers.

- Grid status and control information must be easily available to any actor (participant) at any location.
- Information delivery (both status and control) must be timely and reliable, guaranteeing relevant QoS properties.
- Status and control information must be protected against illegitimate use.

A possible solution to the issues mentioned above is to develop a service-oriented middleware (possibly a composable service architecture presented in this dissertation) built upon underlying network technologies addressing heterogeneity and QoS requirements of the power grid. This enables cooperation of different actors with different capabilities, while heterogeneous entities can discover each other and be executed in coordinated fashion. End-to-end QoS requirements can be assured with combination of the following methodologies: balancing offer and demand, verification of composition correctness, probabilistic and best-effort management. Using service-oriented architecture instead of the existing communication infrastructure further improves interoperability and composability across multiple domains. Finally, actors can dynamically enter and leave the system, or change their properties.

Furthermore, what implicitly happens in the proposed scenario is interplay of different infrastructures: by deploying service oriented middleware for managing power grid (electric power infrastructure) over existing communication channels (telecommunication infrastructure), added value services are being generated in combination with new information infrastructure (e.g., power grid early warning system). Newly generated services are cross-infrastructure services, as they span both power- and telco-infrastructure and must address mutual dependencies of both.

This is not happening in the electric power industry only: deregulation, globalization and dynamic interactions are forcing many legacy paradigms to be rethought with the goal to adopt more flexible and loosely-coupled solutions accommodating the rapidly changing environment. Therefore, a new question arises: should this migration be done on a per-case basis, or is there a need to introduce *service engineering* [163] as a new discipline, or even a new *service science* [138]?

Both service engineering and service science are interdisciplinary in a sense that they represent not only elements of computer science knowledge base (networking, distributed systems, database systems) , but also economy (system analysis, workforce management, business process modeling) and social sciences (ethnography, demography, psychology) knowledge base. The proponents of service science argue that its creation is analogous to the

way computer science was created out of electrical engineering and mathematics some decades ago. Several leading universities are currently seriously evaluating whether to include service science as a part of their curriculum, positioning it between standard engineering and MBA courses.

Although the need for service science may be premature at this point, a common understanding should nevertheless be that the main focus of service-oriented computing should not be only on the existing technologies and simplified examples, but on architectural approaches enabling gradual and seamless entrance into the world of interplay of different infrastructures, where new, added-value and hybrid services are created dynamically, spanning legal, technological and business barriers.

8.3 Future Work

One of the main aspects of the the future work should concentrate on improving the proposed architecture by addressing the following issues:

- Automatic composition
  - Alternative search strategies
  - Optimality
  - Parallel search over multiple directories
- Grid application:
  - Self-repairing and self-healing grids
  - Fault tolerance
- Migration to peer-to-peer environment
- Mobility
  - Application in ad hoc networks
  - Embedded systems
- Guaranteeing real-time behavior
- Ubiquitous and personal networks
- Porting to JAX-WS platform

We are aware that space and time complexity are major limiting factors in the practical application of the automatic composition. Although developed heuristics show promising results in the tests, applications on densely populated directories may result in a performance bottleneck. Therefore, additional search methodologies will be investigated, such as potential application of Iterative-Deepening A* (IDA*) [82], where best-first node expansion is simulated by a series of depth-first searches. Possibility of modeling automatic composition as a game tree search problem will be investigated for the purpose of application of Alpha-Beta search [79], SSS* [159] and its recursive variants [135]. Another important issue is development of plausible scenario test bed that can show the practical advantages and drawbacks of different solutions. A major factor that makes such realization difficult is the necessity to deploy a service directory with adequate number of CDL-annotated services. Returning to the analytical comparison, only the question of feasibility has been addressed in this work. The future work should perform optimality analysis. A separate effort is already under way to prove that the basic heuristic search is optimal if heuristic function given by distance function δ does not overestimate distance to the goal state, as well as to provide analytical dependency between futility value F and heuristic function δ. Additional analysis will be performed in the areas of negotiating suboptimal solutions and finding optimal solution by the given criteria (e.g., price or execution time). Finally, it is expected that search may be performed upon multiple directories. In that case, multiple instances of search algorithm can run on each directory in effort to parallelize the search process. Two problems must be solved in such case: partitioning of state space and ensuring optimal work distribution among directories and algorithm instances [134].

Grid is inherently service-oriented and the ability to dynamically and on-demand replace failed services within the grid has the potential to increase the overall grid fault-tolerance. Automatic composition can be used to achieve this goal, thus enabling self-healing and self-repairing grids. Furthermore, an immediate benefit of composable service architecture application to the service grid infrastructure is the ability to verify the correctness of the user's request. By discarding all incorrect requests, grid performance as well as grid integrity can be improved.

Migration of the proposed solution to the peer-to-peer environment is perceived as an important task. As already discussed in the Section 7.4, such migration is faced with multiple research and practical challenges. Peer-to-peer database design will be explored in order to determine the feasibility of the distributed heterogeneous service directory [143]. An effort to augment Web Service standards will be undertaken with the goal to support direct, peer-to-peer service interaction [144]. The goal is to eliminate the perfor-

mance bottleneck of the composition server and to address fault-tolerance and security issues by eliminating single point of failure. For that purpose, further research in the area of distributed (shared) trust is necessary.

Service-based solutions are becoming increasingly attractive in the area of mobile computing with the promise of seamless connection and cooperation between heterogeneous partners. The issue is particularly relevant in the mobile ad hoc networks (MANETs), where often two very basic premises are assumed: 1) the network is always connected (or that, if partitioned, it will spontaneously regain connectivity after a timeout) and 2) all nodes have the capability of understanding each other for the purpose of either cooperative routing or some other collective task. While the problem of partitioning is being addressed to some extent (e.g., in [117]), almost all MANET solutions still assume that different mobile nodes comprising a typical network will be able to understand each other. This assumption is unrealistic as service-supporting platforms for MANETS are only beginning to appear (e.g., JSR 172 [132] that implements Web Service API for J2ME). Using ideas of composable service architecture in this domain can lead to increased semantic understanding between mobile nodes, enabling them to solve complex tasks, but also to eliminating some security issues through correctness verification. Before service-oriented architecture elements can be deployed in mobile environment, several enabling infrastructural requirements must be ensured, as was demonstrated in [116] for the case of embedded systems.

Most of the today's research in the area of real-time behavior is frequently applied to embedded systems. There exists a need to open this field to the use in heterogeneous and distributed environments. However, guaranteeing timeliness in volatile and best-effort architectures such as SOA presents a major research challenge. Principal problems are unreliable transport protocols, unbounded message delays and worst case execution times. Successful approaches have been made in introducing hard real-time behavior to CORBA using a concept of the composite object [129]. Composable service architecture's contract mechanism, formal composition operators and dynamic service replacement can be used for ensuring real-time behavior for Web Services. It has been shown that composability with respect to real-time can be achieved using the similar approach, albeit in the much more controlled environment [142]. The two open problems of real-time behavior in SOA are that it is very difficult to guarantee timeliness even when a component is executing in the native application server (dependent on server load or other active components), and that communication between services in SOA can result in unbounded message delays. A possible solution that will be investigated is to use as much local knowledge in a more controlled environment (e.g., application domain) as possible and to perform scheduling and dynamic

replacement at the process (composition) level in order to ensure deadlines.

Most user-centric visions of the future information society are focused around the idea that a person is already carrying several intelligent devices with him/her, and that this number will surely climb to many more. Thus the age of pervasive and autonomic computing is entered, where computing resources and infrastructure slowly sink into background and unobtrusively offer their services to the users. This idea is sometimes called *nomadic computing* [78, 77]. In such heterogeneous environment where devices and utilities interact dynamically and on-demand, introducing lightweight and loosely coupled service-oriented architecture as a communication middleware seems a necessity and the only way to manage increasing complexity. Deploying composable service architecture in the ubiquitous context can open the possibility to create diverse personal area network applications, while retaining user's control over the overall environment behavior and effectively subduing somewhat aggressive pervasive computing effects. Hiding the complexity of the underlying computing and communication infrastructure behind the service paradigm can potentially relieve user of configuration, update, maintenance and resource management tasks. Automatic negotiation of quality of service properties and location (context) based services can further improve the user's experience while interacting with the ubiquitous background infrastructure. Furthermore, novel and added value services can be produced in the process of automatic composition, adding to the application quality, fault-tolerance and security. Composability as a property of an architecture further supports ubiquitous visions like NOMADS (Networks of Mobile Adaptive Dependable Systems) Republic that proposes organization of wide area distributed systems according to societal principles [93]. It identifies three key properties: mobility, adaptivity and dependability. Communication and interaction in the NOMADS Republic is performed between citizens that can use and/or provide NOMADS services. Service is defined as an ability to describe and to provide a function in a standard and prescribed way (or being able to answer to W-questions: Who am I, Where am I, What and how can I offer). Enabling correctness verification of service compositions leads to improved dependability, while automatic service composition adds to the adaptiveness property.

Recently, JAX-RPC initiative was promoted and renamed to Java API for XML Web Services (JAX-WS), aiming to alleviate some of the issues plaguing the previous versions [92]. It is planned to investigate the new specification and to port the existing composition server implementation to the new JAX-WS platform. Dissertation contributions are currently being refined and implemented in this context within the frame of EU Integrated

project Adaptive Services Grid (ASG) [3]. The ASG provides a platform, which allows service creation based on next generation Internet technologies. The generic ASG platform consists of a set of languages, concepts, and tools (design and runtime) that are domain independent and support the goal of ASG, i.e., the semantic specification, registration, discovery, composition and enactment of composed and atomic services. The results of this dissertation have found their application in the following activities of the ASG: service discovery and composition, adaptive process management and services grid infrastructure. The concrete elements that are being currently addressed are: revision of non-functional properties, proving and negotiation; contract extraction; additional composition operators; automatic service composition. It is expected that these refinement and implementation efforts will further advance and promote the cause of composability as the property of service-oriented architecture.

SOA is the paradigm that pretends to shape the IT landscape in the following decades. This dissertation introduced composability as necessary and enabling property of the architecture and demonstrated design and implementation benefits archived. Composability is considered to be among deciding factors in accepting SOA and disseminating its effects. The solutions presented in this work may accelerate the process of reorientation towards services, turning it into revolution.

[3] www.asg-platform.org

Bibliography

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1):73–132, 1993.

[2] J.R. Abrial. *The B Book*. Cambridge University Press, 1996.

[3] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S. In *Proceedings of the IEEE Service Computing Conference*, pages 23–30, Shangai, China, 2004.

[4] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9(1):37–49, 1995.

[5] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1987.

[6] S. Alagic and M. Arbib. *The Design of Well Structured and Correct Programs*. Springer Verlag, 1978.

[7] C. Alexander. *A Pattern Language*. Oxford University Press, 1977.

[8] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[9] T. Andrews. Business Process Execution Language for Web Services. *www.ibm.com/ developerworks/library/ws-bpel/*, 2004.

[10] K. Arnout and B. Meyer. Uncovering Hidden Contracts: The .NET Example. *IEEE Computer*, 36(11):48–55, 2003.

[11] F. Baader, D. Calvanese, D. McGuiness, D. Nardi, and P. Patel-Schneider (eds.). *The Description Logic Handbook*. Cambridge University Press, 2003.

[12] H. Balakrishnan, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up Data in P2P Systems. *Communications of the ACM*, 46(2): 43–48, 2003.

[13] R. Baldoni, A. Milani, and S. Tucci Piergiovanni. An Optimal Protocol for Causally Consistent Distributed Shared Memory Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 68–79, Santa Fe, New Mexico, 2004.

[14] R. Baldoni, R. Jiménez-Peris, M. Patiño-Martinez, L. Querzoni, and A. Virgillito. Dynamic Quorums for DHT-based P2P Networks. In *Proceedings of the 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05), Cambridge, MA, USA, July 2005*, 2005.

[15] K. Ballinger, P. Brittenham, A. Malhotra, W.A. Nagy, and S. Pharies. Web Services Inspection Language (WS-Inspection). *http://www-106.ibm.com/developerworks/webservices/ library/ws-wsilspec.html*, 2001.

[16] M. Barborak, M. Malek, and A. Dahbura. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, 25(2):171–220, 1993.

[17] A. Barros and E. Boerger. A Compositional Framework for Service Interaction Patterns and Interaction Flows. In *Proceedings of the Seventh International Conference on Formal Engineering Methods (ICFEM'2005)*, pages 5–35, Manchester, UK, 2005.

[18] A. Barros, M. Dumas, and A. ter Hofstede. Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection. In *Technical Report FIT-TR-2005-02*, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, 2005.

[19] B. Benatallah, M. Dumas, M.-C. Fauvet, and F. Rabhi. Towards Patterns of Web Services Composition. In *Patterns and Skeletons for Parallel and Distributed Computing*, Springer Verlag, 2002.

[20] B. Benatallah, M. Dumas, Q.Z. Sheng, and A.H.H. Ngu. Declarative Composition of Peer-to-Peer Provisioning of Dynamic Web Services. In *Proceedings of the 18th International Conference on Data Engineering (ICDE02)*, pages 297–308, San Jose, USA, 2002.

[21] D. Berardi, D. Calvanese, D. G. Giuseppe, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proceedings of the 1st International Conference on Service Oriented Computing (ICSOC)*, pages 43–58, Trento, Italy, 2003.

[22] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[23] A. Beugnard, J. M. Jezequel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.

[24] G. Booch. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[25] G. Boolos and R. Jeffrey. *Computability and Logic*. Cambridge University Press, 1980.

[26] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *Proceedings of 12th International World Wide Web Conference*, pages 403–410, Budapest, Hungary, 2003.

[27] S. Burbeck. The Tao of e-business Services. *Emerging Technologies, IBM Software Group, ftp://www6.software.ibm.com/software/developer/library/ws-tao.pdf*, 2000.

[28] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[29] L. F. Cabrera. Web Services Coordination. *ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf*, 2004.

[30] J. Cardoso and A.P. Seth. Semantic E-Workflow Composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.

[31] Q. Chen and M. Hsu. Inter-Enterprise Collaborative Business Process Management. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 253–260, Heidelberg, Germany, 2001.

[32] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl*, 2001.

[33] L. Clement, A. Hately, C. Riegen, and T. Rogers (Eds.). UDDI Version 3.0.2. *http://uddi.org/ pubs/uddi-v3.0.2-20041019.pdf*, 2004.

[34] Semantics Web Services Language (SWSL) committee. OWL-S 1.1 Release. *http://www. daml.org/services/owl-s/1.1/*, 2005.

[35] P. M. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. In *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, pages 1–12, Rochester, NY, 1977.

[36] F. Cristian. *Dependability of Resilitent Computers*. Blackwell Scientific Publication, 1989.

[37] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The Next Step in Web Services. *Communications of the ACM*, 46(10):29–34, 2003.

[38] S. Damodaran. B2B Integration over the Internet with XML-RosettaNet Successes and Challenges. In *Proceedings of the 13th international conference on World Wide Web (WWW04)*, pages 188 – 195, New York, USA, 2004.

[39] O. Enseling. iContract: Desing by Contract in Java. *http://www.javaworld.com/javaworld/ jw-02-2001/jw-0216-cooltools.html*, 2001.

[40] G.W. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, New York, 1969.

[41] M. Ernst, W. Griswold, Y. Kataoka, and D. Notkin. Dynamically Discovering Program Invariants Involving Collections. *Technical Report, Univ. of Washington*, 2000.

[42] A. Ankolekar et al. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 348–363, Sardinia, Italy, 2002.

[43] I. Foster et al. Modeling stateful resources with web services. *http://www.ibm.com/ developerworks/library/ws-resource/ws-modelingresources.pdf*, 2004.

[44] K. Czajkowski et al. The WS-Resource Framework. *http://www.globus.org/wsrf/specs/ws-wsrf.pdf*, 2004.

[45] R.J. Bayardo et al. InfoSleuth: Agent-based Semantic Integration of Information in Open and Dynamic Environments. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 195 – 206, New York, USA, 1997.

[46] D. Fahland and W. Reisig. ASM-based semantics for BPEL: The negative Control Flow. In *roceedings of the 12th International Workshop on Abstract State Machines (ASM'05)*, pages 131–151, Paris, France, 2005.

[47] R. Farahbod, Uwe Glasser, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. In *SFU-CMPT-TR-2004-03*, Technial report, Simon Fraser University, Canada, 2004.

[48] R. T. Fielding. *Architectural Styles and the Design of Network-Based Software Architectures*. doctoral dissertation, Dept. of Computer Science, Univ. of California, Irvine, 2000.

[49] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Physiology of the Grid. *http://www.globus.org/alliance/publications/papers/ogsa.pdf*, 2005.

[50] W. Fouché. Arithmetical representations of Brownian motion. *Journal of Symbolic Logic*, 65(1):421–442, 2002.

[51] X. Fu, T. Bultan, and J. Su. Formal Verification of E-Services and Workflows. In *Proceedings of Workshop on Web Services, e-Business, and the Semantic Web (WES): Foundations, Models, Architecture, Engineering and Applications*, pages 188–202, Toronto, Canada, 2002.

[52] X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA)*, pages 188–200, Santa Barbara, USA, 2003.

[53] E. Gamma, R. Helm, R. Johnson, and J. Ulissides. *Design Patterns*. Addison-Wesley, 1995.

[54] K. Garlington. Critique of 'Put it in the contract: The lessons of Ariane'. *http://home.flash. net/˜kennieg/ariane.html*, 1997.

[55] J. Gaschnig. *Performace Measurement and Analysis of Certain Search Algorithms*. Doctoral dissertation, Carnegie-Mellon University, 1979.

[56] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning*. Morgan Kauffman, 2004.

[57] S. Ghandeharizadeh, C.A. Knoblock, C. Papadopoulos, C. Shahabi, E. Alwagait, J.L. Ambite, M. Cai, C. Chen, P. Pol, R. Schmidt, S. Song, S. Thakkar, and R. Zhou. Proteus: A System for Dynamically Composing and Intelligently Executing Web Services. In *Proceedings of the 2003 International Conference on Web Services (ICWS'03)*, pages 17–21, Las Vegas, USA, 2003.

[58] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[59] D. Gries. *The Science of Programming*. Springer Verlag, 1987.

[60] Object Management Group. Technical Guide to Model Driven Architecture: The MDA Guide. *http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf*, 2003.

[61] M. Gudgin, M. Hadley, N. Mendelsohn, and H. F. Nielsen J-J. Moreau. SOAP Version 1.2. *http://www.w3.org/TR/soap12-part1/*, 2003.

[62] Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In *Current Trends in Theoretical Computer Science*. World Scientific, 1993.

[63] R. Hamadi and B. Benatallah. A Petri Net-based model for Web Service Composition. In *Proceedings of the 14th Australasian database conference on Database technologies*, pages 191–200, Adelaide, Australia, 2003.

[64] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions Software Engineering and Methodology*, 5(4):293–333, 1996.

[65] P.E. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[66] P.E. Hart, N.J. Nilsson, and B. Raphael. Correction to 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths'. *SIGART Newsletter*, Vol 37:28–29, 1972.

[67] C.H. Hauser, D.E. Bakken, and A. Bose. A Failure to Communicate. *IEEE Power and Energy*, pages 10–18, March/April 2005.

[68] S. Hinz, K. Schmidt, and Ch. Stahl. Transforming BPEL to Petri Nets. In *Proceedings of Third International Conference on Business Process Management (BPM 2005)*, pages 220–235, Nancy, France, 2005.

[69] P. Horn. The New Discipline of Services Science. In *http://www.businessweek.com/ technology/content/jan2005/tc20050121_ 8020.htm*, Business Week Online, 2005.

[70] A. Igarashi and N. Kobayashi. A generic type system for lock-free processes. In *Proceedings of 26th ACM Symposium on Principles of Programming Languages*, pages 128–141, London, United Kingdom, 2001.

[71] W. Iverson. *Real World Web Services*. O'Reilly, 2004.

[72] J. M. Jezequel and B. Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

[73] H. Karloff. *Linear Programming*. Birkhauser, 1991.

[74] A. H. Karp. E-speak Explained. *Technical Report, Hewllet-Packard Laboratories HPL-2000-101*, 2000.

[75] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language Version 1.0. In *http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/*, 2004.

[76] M. Kiefer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the Association for Computing Machinery (ACM)*, 42(4):741 – 843, 1995.

[77] L. Kleinrock. Nomadic computing - an opportunity. *ACM Computer Communication Review*, 25(1):36–40, 1995.

[78] L. Kleinrock. Nomadic computing. In *Procceedings of the International Conference on Mobile Computing and Networking, Berkeley, Ca*, 1995.

[79] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[80] P. Kollock. The production of trust in online markets. *Advances in Group Processes*, Vol. 16, 1999.

[81] H. Kopetz and N. Suri. On the Limits of the Precise Specification of Component Interfaces. In *Proceedings of the 9th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS2003F)*, 2003.

[82] R.E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[83] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: Service-Oriented Architecture Best Practices (The Coad Series).* Prentice Hall PTR, 2004.

[84] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[85] P.A. Lee and T. Anderson. Fault Tolerance Principles and Practice. In *Dependable Computing and Fault Tolerant Systems, Volume 3*, Springer-Verlag, 1990.

[86] Maren Lenk. *Heuristic Composition of Abstract Machines.* Master thesis, Institute for Informatics, Humboldt University Berlin, 2005.

[87] M. Leuschel. The ProB Animator and Model Checker for the B Method. *http://www.ecs. soton.ac.uk/~mal/systems/prob.html*, 2005.

[88] M. Leuschel and M. Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[89] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall, 2000.

[90] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proceedings of the 12th international conference on World Wide Web*, pages 331 – 339, BudBudapest, Hungary, 2003.

[91] Q.A. Liang and S.Y.W. Su. AND/OR Graph and Search Algorithm for Discovering Composite Web Services. *International Journal of Web Services Research*, 2(4):48–67, 2005.

[92] S. Loughran and E. Smith. Rethinking the Java SOAP Stack. *HP Laboratories Bristol Technical Report, HPL-2005-83*, 2005.

[93] M. Malek. Introduction to NOMADS. In *Proceedings of the Computing Frontiers*, pages 26–27, Ischia, Italy, 2004.

[94] F. Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Wiley, 2002.

[95] A. Martelli and U. Montanari. Additive And/Or Graphs. In *Proceedings of the 3rd International Joint Confrence on Artificial Intelligence*, pages 1–7, Stanford, USA, 1973.

[96] A. Martelli and U. Montanari. Optimization Decision Trees Through Heuristically Guided Search. *Communication of the ACM*, 21(12): 1025–1039, 1978.

[97] S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the International Conference on the Principles of Knowledge Representation and Reasoning (KRR'02)*, pages 482–496, Tolouse, France, 2002.

[98] L.G. Meredith and S. Bjorg. Contracts and Types. *Communications of the ACM*, 46(10):41–47, 2003.

[99] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[100] B. Meyer. Contracts for Components. *Software Development*, 8(7): 51–53, 2000.

[101] B. Meyer. Applying Design by Contract. *IEEE Computer*, 25(10): 40–51, 1992.

[102] B. Meyer. Towards Practical Proofs of Class Correctness. In *Proceedings of ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users*, pages 359–387, Turku, Finland, 2003.

[103] Sun Microsystems. Javadoc Tool Home Page. *http://java.sun.com/j2se/javadoc/*, 2005.

[104] Sun Microsystems. Jini Technology Core Platform Specification. *http://www.sun.com/ software/jini/specs/core2_ 0.pdf*, 2004.

[105] Sun Microsystems. The Java Web Services Developer Pack. *http://java.sun.com/ webservices/downloads/webservicespack.html*, 2005.

[106] Sun Microsystems. Java pet store. *http://java.sun.com/developer/releases/petstore/*, 2005.

[107] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous web services. In *Proceedings of the Workshop of Dependable Middleware-based Systems*, pages 44–53, Washington D.C., USA, 2002.

[108] N. Milanovic. Contract-based Web Service Composition Framework with Correctness Guarantees. In *Proceedings of the 2nd International Service Availability Forum (ISAS)*, pages 46–59, Berlin, Germany, 2005.

[109] N. Milanovic and M. Malek. Extracting Functional and Non-functional Contracts from Java Classes and Enterprise Java Beans. In *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2004) at the International Conference on Dependable Systems and Networks (DSN 2004)*, pages 282–286, Florence, Italy, 2004.

[110] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[111] N. Milanovic and M. Malek. Verifying Correctness of Web Services Composition. In *Proceedings of the 11th Infofest*, pages 219–231, Budva, Montenegro, 2004.

[112] N. Milanovic and M. Malek. Search Strategies for Automatic Web Service Composition. *International Journal of Web Services Research*, 3(2):1–32, 2006.

[113] N. Milanovic and M. Malek. Architectural Support for Automatic Service Composition. In *Proceedings of the IEEE Service Computing Conference (SCC 2005)*, pages 133–140, Orlando, USA, 2005.

[114] N. Milanovic, V. Stantchev, J. Richling, and M. Malek. Towards Adaptive and Composable Services. In *Proceedings of the International IPSI2003 Conference*, Sveti Stefan, Montenegro, 2003.

[115] N. Milanovic, M. Malek, A. Davidson, and V. Milutinovic. Routing and Security in Mobile Ad Hoc Networks. *IEEE Computer*, 37(2):61–65, 2004.

[116] N. Milanovic, J. Richling, and M. Malek. Lightweight Services for Embedded Systems. In *Proceedings of the 2nd IEEE Workshop on Software Technologies for Embedded and Ubiquitous Computing Systems (WST-FEUS 2004)*, pages 40–44, Vienna, Austria, 2004.

[117] B. Milic, N. Milanovic, and M.Malek. Prediction of Partitioning in Location-aware Mobile Ad Hoc Networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-38*, pages 306–312, Hawaii, USA, 2005.

[118] C. Mills. Using Design by Contract in C. *http://www.onlamp.com/pub/a/onlamp/2004/10/28/ design_ by_ contract_ in_ c.html*, 2005.

[119] R. Milner. The Polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer Verlag, 2003.

[120] M.P.Papazoglou and D. Georgakopoulos. Service Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.

[121] N. Medvidovic N and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[122] S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services . In *Proceedings of the 11th International World Wide Web Conference*, pages 77–88, Honolulu, Hawaii, USA, 2002.

[123] A. Naumenko, S. Nikitin, V. Terziyan, and J. Veijalainen. Using UDDI for Publishing Metadata of the Semantic Web. In *Proceedings of the First International IFIP/WG12.5 Working Conference on Industrial Applications of Semantic Web*, pages 84–98, Jyvaskyla, Finland, 2005.

[124] A. Newell, J.C. Shaw, and H.A. Simon. Empirical Explorations with the Logic Theory Machine: A Case Study in Heuristics. In *Computers and Thought*, McGraw-Hill, New York, 1963.

[125] N.J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[126] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347, Sardinia, Italy, 2002.

[127] M. Paolucci, T. Kawamura, and T.R. Payne K. Sycara. Importing the Semantic Web in UDDI. In *Proceedings of Web Services, E-Business and Semantic Web Workshop, CAiSE*, pages 225–236, Toronto, Canada, 2002.

[128] A. Pneuli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, New York, USA, 1977.

[129] A. Polze, J. Richling, J. Schwarz, and M. Malek. Towards Predictable CORBA-based Web-Services. In *Proceedings of 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing*, pages 182–191, St.Malo, France, 1999.

[130] D. Powell. Automatic derivation of Loop Termination Conditions to Support Verification. In *27th Australasian Computer Science Conference*, pages 89–97, Dunedin, New Zealand, 2004.

[131] G. Prasad, R. Taneja, and V. Todankar. Web and Enterprise Architecture Design Patterns for J2EE. *O'Reilly OnJava, http://www.onjava.com/lpt/a/4161*, 2003.

[132] Java Community Process. JSR-00172 J2ME Web Services Specification, 2006.

[133] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, 1996.

[134] A. Reinefeld. Scalability of Massively Parallel Depth-First Search. In *Parallel Processing of Discrete Optimization Problems*, volume 22, pages 305–322, ACM Press, DIMACS Series in Discrete Mathem. and Theor. Comp. Sc., 1995.

[135] A. Reinefeld and P. Ridinger. Time-efficient state space search. *Artificial Intelligence*, 71(2):397–408, 1994.

[136] A. Reinefeld and F. Schintke. Grid Services. *Informatik-Spektrum*, 27(2):129–135, 2004.

[137] A. Reinefeld and F. Schintke. Concepts and Technologies for a Worldwide Grid Infrastructure. In *Euro-Par 2002 Parallel Processing*, pages 62–71, Springer LNCS 2400, 2002.

[138] IBM Research. Services Science: A New Academic Discipline. In *http://domino.research.ibm.com/comm/www_fs.nsf/images/fsr/$FILE/summit_report.pdf*, 2005.

[139] Microsoft Research. Reference Manual of AsmL. In *http://research.microsoft.com/fse/asml/ doc/AsmL2_Reference.doc*, 2005.

[140] P. Resnick, R. Zeckhauser, E. Friedman, and K. Kuwabara. Reputation Systems. *Communications of the ACM*, 43(12):45–48, 2000.

[141] E. Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[142] J. Richling. *Komponierbarkeit eingebetteter Echtzeitsysteme*. PhD thesis, Humboldt University, Berlin, 2006.

[143] T. Risse and P. Knezevic. Data Storage Requirements for the Service Oriented Computing. In *Proceedings of the IEEE Workshop on Service Oriented Computing*, pages 67–72, Los Alamitos, USA, 2003.

[144] T. Risse, P. Knezevic, and A. Wombacher. P2P Evolution: From Filesharing to Decentralized Workflows. *Information Technology*, pages 193–199, 2004.

[145] L. Rodrigues, R. Baldoni, E. Anceaume, and M. Raynal. Deadline-constrained Causal Order. In *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC)*, pages 234–241, 2000.

[146] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.

[147] E. Roman. *Mastering Enterprise Java Beans*. Wiley Computer Publishing, 2002.

[148] RosettaNet. *http://www.rosettanet.org/*, 2006.

[149] M. Sabou, C. Wroe, C. Goble, and G. Mishne. Learning Domain Ontologies for Web Service Descriptions: an Experiment in Bioinformatics. In *Proceedings of the International Conference on World Wide Web (WWW2005)*, pages 190 – 198, Chiba, Japan, 2005.

[150] B.-H. Schlingloff, A. Martens, and K. Schmidt. Modeling and Model Checking Web Services. *Electronic Notes in Theoretical Computer Science: Issue on Logic and Communication in Multi-Agent Systems*, 126: 3–26, 2005.

[151] H. Schmidt, I. Poernomo, and R. Reussner. Trust-by-Contract: Modelling, analysing and predicting behaviour of software architectures. *Journal of Integrated Design and Process Science*, 5(3):25–51, 2001.

[152] Q.Z. Sheng, B. Benatallah, M. Dumas, and E.O.Y Mak. SELF-SERV: A Platform for Rapid Composition of Web Services in Peer-to-Peer Environment. In *Proceedings of the 28th Very Large Databases Conference (VLDB)*, pages 1051–1054, Hong Kong, China, 2002.

[153] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic Composition of Web Services using Semantic Descriptions. In *Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS 2003*, pages 17–24, Angers, France, April 2003.

[154] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[155] J.M. Snell. Web services programming tips and tricks: Learn simple, practical Web services design patterns. *www-106.ibm.com/developerworks/library/ws-tip-altdesign1/, /ws-tip-altdesign2/, /ws-tip-altdesign3/, /ws-tip-altdesign4/*, 2005.

[156] Eiffel Soft. The Home of EiffelStudio and Eiffel ENVision. *www.eiffel.com*, 2004.

[157] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2001.

[158] L.A. Stenvold, J. Grav, and S. Bergvik. User Experiences of Work Group Awareness Information Provided By a Buddy List Application. *Telenor Research and Development*, report TFoU R21/99, 1999.

[159] G.C. Stockman. A minimax algorihtm faster than alpha-beta. *Artificial Intelligence*, 12(2):179–196, 1979.

[160] K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace. In *Autonomous Agents and Multi-Agent Systems*, pages 173 – 203. Kluwer Academic Press, 2002.

[161] F. Tartanoglu, V. Issarny, and A. Romanovsky. Dependability in the Web Service Architecture. In *Architecting Dependable Systems*, LNCS 2677, Springer-Verlag, 2003.

[162] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy. Coordinated forward error recovery for composite web services. In *Proceeding of the 22nd International Symposium on Reliable Dependable Systems, SRDS 2003*, pages 167–176, Florence, Italy, 2003.

[163] J.M. Tien and D. Berg. A Case for Service Systems Engineering. *Journal of Systems Science and Engineering*, 12(1):13–38, 2003.

[164] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 2(42), pages 230–265, 1936.

[165] W.M.P. vad der Aalst and Akhil Kumar. XML based schema Definition for Support of Inter-organizational Workflow. *Information Systems Research*, 14(1):23–46, 2003.

[166] W.J. van den Heuvel and Z. Maamar. Moving Toward a Framework to Compose Intelligent Web Services. *Communications of the ACM*, 46(10):103–109, 2003.

[167] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems Research*, 30(4):245–275, 2005.

[168] A. van Moorsel. On Best-Effort and Dependability. In *Proceedings of the 2nd International Service Availability Forum (ISAS)*, pages 99–101, Berlin, Germany, 2005.

[169] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management*, 6(1):17–39, 2004.

[170] S. Vinoski. WS-Nonexistent Standards. *IEEE Internet Computing*, 8(6):25–28, 2004.

[171] W. Vogels. Web Services are not Distributed Objects: Common Misconceptions about the Fundamentals of Web Service Technology. *IEEE Internet Computing*, 7(6):59–66, 2003.

[172] W3C. Resource Description Framework (RDF). *http://www.w3.org/RDF/*, 2006.

[173] W3C. Web Services Architecture. *http://www.w3.org/TR/ws-arch/*, 2004.

[174] J. Webber and S. Parastatidis. Demystifying Service-Oriented Architecture. *Web Services Journal*, 3(11), November 2003.

[175] J. Webber, S. Parastatidis, and M. Little. Stateful Interactions in Web Services: A Comparison of WS-Context and WS-Resource Framework. *Web Services Journal*, 4(6), 2004.

[176] M. Werner and J. Richling. Komponierbarkeit nichtfunktionaler Eigenschaften - Versuch einer Definition (engl: Composability of non-functional properties — an attempt of a definition). In *GI Fachtagung Betriebssysteme*, Berlin, 2002.

[177] M. Werner, J. Richling, N. Milanovic, and Vladimir Stantchev. Composability Concept for Dependable Embedded Systems. In *Proceedings of the International Workshop on Dependable Embedded Systems in conjuction with SRDS 2003*, pages 20–25, Florence, Italy, 2003.

[178] D.B. West. *Introduction to Graph Theory*. Prentice Hall, 1996.

[179] E. J. Weyuker. Testing Component Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, 1988.

[180] ESSI WSMO working group. Web Service Modeling Ontology. *http://wsmo.org/*, 2006.

[181] SDK WSML working group. The Web Service Modeling Language WSML. *http://www. wsmo.org/wsml/wsml-syntax/*, 2006.

[182] J. Yang. Web Service Componentization. *Communications of the ACM*, 46(10):35–40, 2003.

[183] J. Yang and M. P. Papazoglou. Web Component: A Substrate for Web Service Reuse and Composition. In *Proceedings of 14th Conference on Advanced Information Systems Engineering (CAiSE02)*, pages 21–36, Toronto, Canada, 2002.

[184] J. Yang and M.P. Papazoglou. Interoperation Support for electronic business. *Communications of the ACM*, 43(6):39–47, June 2000.

[185] N. Yoshida, K. Honda, and M. Berger. Linearity and Bisimulation. In *Proceedings of the Fifth International Conference, Foundations of Software Science and Computer Structures*, pages 417–434, Grenoble, France, 2002.

[186] N. Yoshikai, H. Takahashi, and Y. Usui. Experimental evaluation of reputation systems on internet auctions. In *Proceedings of the International SSGRR2003s Conference*, L'Aquila, Italy, 2003.

[187] L. Zeng. *Dynamic Web Services Composition.* PhD thesis, University of New South Wales, 2003.

[188] L. Zeng, B. Benatallah, and A.H.H Ngu. On Demand Business-to-Business Integration. In *Proceedings of the 9th International Conference Cooperative Information Systems*, pages 403 – 417, Trento, Italy, 2001.

[189] L. Zeng, B. Benatallah, M. Dumas, and J. Kalagnanam. Quality Driven Web Services Composition. In *Proceedings of the 12th International Conference World Wide Web*, pages 411 – 421, Budapest, Hungary, 2003.

[190] L. Zeng, B. Benatallah, A.H.H Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions of Software Engineering*, 30(5):311–327, 2004.

[191] J. Zhang. Trustworthy Web Services: Actions for Now. *IEEE IT Professional*, 7(1):32–36, 2005.

[192] J. Zhang, C.K. Chang, J.Y. Chung, and S.W. Kim. S-Net: A Petri-net Based Specification Model for Web Services. In *Proceedings of IEEE International Conference on Web Services*, pages 420–427, San Diego, USA, 2004.

Appendix A

Contract Definition Language XSD Schema

```
<?xml version="1.0" encoding="utf-8"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">CDL Schema</xsd:documentation>
   </xsd:annotation>
   <xsd:element name="contract" type="contractType"/>
   <xsd:complexType name="contractType">
     <xsd:sequence>
       <xsd:element name="organization" type="organizationType"/>
       <xsd:element name="types" type="typesType"/>
       <xsd:element name="location" type="locationType" minOccurs="0"/>
       <xsd:element name="method" type="methodType" maxOccurs="unbounded"/>
       <xsd:element name="event" type="eventType" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="serviceURI" type="xsd:string" use="required"/>
     <xsd:attribute name="serviceName" type="xsd:string" use="required"/>
     <xsd:attribute name="serviceDescription" type="xsd:string" use="optional"/>
     <xsd:attribute name="price" type="xsd:decimal" use="optional"/>
     <xsd:attribute name="state" type="xsd:string" use="optional" default="stateless"/>
     <xsd:attribute name="version" type="xsd:string" use="optional"/>
     <xsd:attribute name="port" type="xsd:string" use="required"/>
   </xsd:complexType>
   <xsd:complexType name="typesType">
     <xsd:sequence>
       <xsd:element name="targetNamespace" type="xsd:string"/>
       <xsd:element name="complexType" type="complexTypeType" maxOccurs="unbounded"/>
     </xsd:sequence>
   </xsd:complexType>
   <xsd:complexType name="complexTypeType">
     <xsd:sequence>
       <xsd:element name="sequence">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="element" maxOccurs="unbounded">
               <xsd:complexType>
                 <xsd:attribute name="name" type="xsd:string"/>
                 <xsd:attribute name="type" type="xsd:string"/>
               </xsd:complexType>
             </xsd:element>
           </xsd:sequence>
```

```
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="organizationType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="description" type="xsd:string" minOccurs="0"/>
    <xsd:element name="classification" type="classificationType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="primaryContact" type="primaryContactType"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="locationType">
  <xsd:sequence>
    <xsd:element name="country" type="xsd:string" minOccurs="0"/>
    <xsd:element name="city" type="xsd:string" minOccurs="0"/>
    <xsd:element name="street" type="xsd:string" minOccurs="0"/>
    <xsd:element name="GPS" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="latitude" type="xsd:string"/>
          <xsd:element name="longitude" type="xsd:string"/>
          <xsd:element name="height" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="classificationType">
  <xsd:sequence>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="value" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="primaryContactType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="phone" type="xsd:string"/>
    <xsd:element name="email" type="xsd:string" minOccurs="0"/>
    <xsd:element name="address" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="methodType">
  <xsd:sequence>
    <xsd:element name="parameters" type="parameterType"/>
    <xsd:element name="resource" type="resourceType" minOccurs="0"/>
    <xsd:element name="B-spec" type="xsd:string" minOccurs="0"/>
    <xsd:element name="invocation" type="invocationType" minOccurs="0"/>
    <xsd:element name="precondition" type="preconditionType" minOccurs="0"/>
    <xsd:element name="postcondition" type="postconditionType" minOccurs="0"/>
    <xsd:element name="invariant" type="invariantType" minOccurs="0"/>
    <xsd:element name="event-ref" type="xsd:string" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="assertions" type="xsd:string" minOccurs="0"/>
    <xsd:element name="classification" type="classificationType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="location" type="locationType" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
  <xsd:attribute name="uri" type="xsd:string" use="required"/>
```

```
  <xsd:attribute name="port" type="xsd:string" use="required"/>
  <xsd:attribute name="description" type="xsd:string"/>
  <xsd:attribute name="price" type="xsd:decimal"/>
  <xsd:attribute name="version" type="xsd:string"/>
</xsd:complexType>
<xsd:complexType name="resourceType">
  <xsd:sequence>
    <xsd:element name="resourceID">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="resourceName" type="xsd:string"/>
          <xsd:element name="resourceURI" type="xsd:string"/>
          <xsd:element name="resourceManager" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="resourceAction" type="xsd:string"/>
    <xsd:element name="resourceProperty" type="xsd:string"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="parameterType">
  <xsd:sequence>
    <xsd:element name="set" type="setType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="constants" type="constantType" minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="param" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="type" type="xsd:string"/>
          <xsd:element name="restriction" type="xsd:string" minOccurs="0"/>
          <xsd:element name="initialization" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="direction" type="xsd:string" use="required"/>
        <xsd:attribute name="requiredParam" type="xsd:string"/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="setType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="domain" type="xsd:string"/>
    <xsd:element name="restriction" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="constantType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="value" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="invocationType">
  <xsd:sequence>
    <xsd:element name="create" type="callType" minOccurs="0"/>
    <xsd:element name="message" type="messageType" minOccurs="0"/>
    <xsd:element name="time" minOccurs="0">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="SYNC|ASYNC"/>
```

```
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="type">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="SOAP"/>
        <xsd:enumeration value="RMI-IIOP"/>
        <xsd:enumeration value="NET-REM"/>
        <xsd:enumeration value="JAXRPC"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="messageType">
  <xsd:sequence>
    <xsd:element name="channel" type="xsd:string" minOccurs="0"/>
    <xsd:element name="producer" type="xsd:string" minOccurs="0"/>
    <xsd:element name="consumer" type="xsd:string" minOccurs="0"/>
    <xsd:element name="duration" type="xsd:duration" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="callType">
  <xsd:sequence>
    <xsd:element name="home" type="xsd:string" minOccurs="0"/>
    <xsd:element name="remote" type="xsd:string" minOccurs="0"/>
    <xsd:element name="local" type="xsd:string" minOccurs="0"/>
    <xsd:element name="factory" type="xsd:string" minOccurs="0"/>
    <xsd:element name="port" type="xsd:string" minOccurs="0"/>
    <xsd:element name="context" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="clauseType">
  <xsd:sequence>
    <xsd:element name="render" type="renderType" minOccurs="0"/>
    <xsd:element name="log" type="logType" minOccurs="0"/>
    <xsd:element name="security" type="securityType" minOccurs="0"/>
    <xsd:element name="dependability" type="dependabilityType"
      minOccurs="0"/>
    <xsd:element name="performance" type="performanceType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="params" type="xsd:string" minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="preconditionType">
  <xsd:complexContent><xsd:extension base="clauseType"/></xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="postconditionType">
  <xsd:complexContent><xsd:extension base="clauseType"/></xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="invariantType">
  <xsd:complexContent><xsd:extension base="clauseType"/></xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="securityType">
  <xsd:sequence>
    <xsd:element name="authentication" type="authenticationType"
      minOccurs="0"/>
    <xsd:element name="authorization" type="authorizationType"
      minOccurs="0"/>
  </xsd:sequence>
```

```
</xsd:complexType>
<xsd:complexType name="authenticationType">
  <xsd:sequence>
    <xsd:element name="credential_ref" type="xsd:string"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="authorizationType">
  <xsd:sequence>
    <xsd:element name="security_role" type="xsd:string"
      maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="dependabilityType">
  <xsd:sequence>
    <xsd:element name="transactions" type="transactionType" minOccurs="0"/>
    <xsd:element name="replication" type="replicationType" minOccurs="0"/>
    <xsd:element name="check-point" type="checkpointType" minOccurs="0"/>
    <xsd:element name="timeout" minOccurs="0">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:string">
            <xsd:attribute name="unit" type="xsd:string"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="exceptions" type="exceptionType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="transactionType">
  <xsd:sequence>
    <xsd:element name="transaction-manager" type="xsd:string" minOccurs="0"/>
    <xsd:element name="resource" type="xsd:string" minOccurs="0"/>
    <xsd:element name="resource-manager" type="xsd:string" minOccurs="0"/>
    <xsd:element name="compensate-method" type="xsd:string" minOccurs="0"/>
    <xsd:element name="timeout" minOccurs="0">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:duration">
            <xsd:attribute name="unit" type="xsd:string" use="required"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="transaction-context" type="xsd:string" minOccurs="0"/>
    <xsd:element name="enlist" minOccurs="0">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="not-supported"/>
          <xsd:enumeration value="supports"/>
          <xsd:enumeration value="required"/>
          <xsd:enumeration value="requires-new"/>
          <xsd:enumeration value="mandatory"/>
          <xsd:enumeration value="never"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="isolation" minOccurs="0">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="read-uncommitted"/>
```

```
          <xsd:enumeration value="read-committed"/>
          <xsd:enumeration value="repeatable-read"/>
          <xsd:enumeration value="serializable"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="model" use="optional">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="FLAT|NESTED"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
<xsd:complexType name="replicationType">
  <xsd:sequence>
    <xsd:element name="number-of-copies" type="xsd:integer" minOccurs="0"/>
    <xsd:element name="addressing" type="xsd:string" minOccurs="0"/>
    <xsd:element name="broadcast-type" type="xsd:string" minOccurs="0"/>
    <xsd:element name="ordering" type="xsd:string" minOccurs="0"/>
    <xsd:element name="delivery" type="xsd:string" minOccurs="0"/>
    <xsd:element name="response" type="xsd:string" minOccurs="0"/>
    <xsd:element name="group-structure" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="checkpointType">
  <xsd:sequence>
    <xsd:element name="frequency" minOccurs="0">
      <xsd:complexType>
        <xsd:simpleContent>
          <xsd:extension base="xsd:duration">
            <xsd:attribute name="unit" type="xsd:string" use="required"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="outfile" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="exceptionType">
  <xsd:sequence>
    <xsd:element name="application" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="exc" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="native" type="xsd:string"/>
                <xsd:element name="generic" type="xsd:string"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="system" minOccurs="0">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="exc" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:sequence>
```

```
                <xsd:element name="native" type="xsd:string"/>
                <xsd:element name="generic" type="xsd:string"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="performanceType">
  <xsd:sequence>
    <xsd:element name="type" type="xsd:string"/>
    <xsd:element name="unit" type="xsd:string"/>
    <xsd:element name="value" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="logType">
  <xsd:sequence>
    <xsd:element name="location">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="security" type="xsd:string" minOccurs="0"/>
          <xsd:element name="traffic" type="xsd:string" minOccurs="0"/>
          <xsd:element name="system" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="eventType">
  <xsd:sequence>
    <xsd:element name="reference" type="xsd:string"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="wrapper" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Appendix B

Abstract Machine Notation

B.1 Non-freeness

A variable has a free occurrence in a predicate or in an expression if it is present in such formula and it is present in a sub-formula not under scope of an universal or existential quantifier. Conversely, a variable has no free occurrences in a predicate or in an expression if it is not present in such formula, or it is only present in sub-formulae which are under the scope of universal or existential quantifiers that introduce that variable as their quantified variable. For example, in the following formula:

$$\forall p \exists q (p \in R \Rightarrow p + q < x)$$

Variables p, q and r have no free occurrences in this formula, because p and q are under the scope of quantifiers, while r does not appear at all. On the other side, variable x is free. For a given variable p and formula Q, non-freeness is expressed with $p \backslash Q$ which reads "variable p is non-free in formula Q". Following are the rules for checking non-freeness, which is important when calculating substitutions for abstract machine operations. Assume that x and y are variables, P and Q are predicates, F is a formula and E is an expression:

$$x \backslash y \longrightarrow x \textit{ and } y \textit{ are distinct}$$
$$x \backslash (P \wedge Q) \longrightarrow x \backslash P \textit{ and } x \backslash Q$$
$$x \backslash (P \Longrightarrow Q) \longrightarrow x \backslash P \textit{ and } x \backslash Q$$
$$x \backslash \neg P \longrightarrow x \backslash P$$
$$x \backslash \forall x \cdot P \longrightarrow \textit{always}$$

$$x \backslash \forall y \cdot P \longrightarrow x \backslash y \ \textit{and}\ x \backslash P$$
$$x \backslash [x := E]F \longrightarrow x \backslash E$$
$$x \backslash [y := E]F \longrightarrow x \backslash y \ \textit{and}\ x \backslash E \ \textit{and}\ x \backslash F$$

In the last two cases, we use non-freeness in substitution ($[x := E]F$), which is covered in Appendix B.2.

B.2 Substitution

Let x be a variable, E an expression, and P a formula, then:

$$[x := E]P$$

denotes substitution that is performed by replacing all free occurrences of x in P by E. Assume that x and y are variables, P and Q are predicates, D and E are expressions, F is a formula. Substitution for different cases is performed as follows:

$$[x := E]x \longrightarrow E$$
$$[x := E]y \longrightarrow y, \ \textit{if}\ x \backslash y$$
$$[x := E](P \wedge Q) \longrightarrow [x := E]P \wedge [x := E]Q$$
$$[x := E](P \Longrightarrow Q) \longrightarrow [x := E]P \Longrightarrow [x := E]Q$$
$$[x := E]\neg P \longrightarrow \neg [x := E]P$$
$$[x := E]\forall x \cdot P \longrightarrow \forall x \cdot P$$
$$[x := E]\forall y \cdot P \longrightarrow \forall y \cdot [x := E]P \ \textit{if}\ y \backslash x \ \textit{and}\ y \backslash E$$
$$[x := x]F \longrightarrow F$$
$$[x := E]F \longrightarrow F \ \textit{if}\ x \backslash F$$
$$[y := E][x := y]F \longrightarrow [x := E]F \ \textit{if}\ y \backslash F$$
$$[x := D][y := E]F \longrightarrow [y := [x := D]E][x := D]F \ \textit{if}\ y \backslash D$$

B.3 One Point Rule

The One Point Rule is a general law of predicate calculus that gives connection between equality and substitution. For a given variable x, expression E and predicate P, the following holds if x has no free occurrences in E.

$$\forall x \cdot (x = E \Longrightarrow P) \Longleftrightarrow [x := E]P \tag{B.1}$$

We prove One Point rule in two lemmas.

Lemma 1 $\forall x \cdot (x = E \Longrightarrow P) \Longrightarrow [x := E]P,$ if $x \backslash E$

Proof:
Let us assume:

$$\forall x \cdot (x = E \Longrightarrow P) \tag{B.2}$$

Then we have to prove:

$$[x := E]P \tag{B.3}$$

under the following assumption:

$$x \backslash E \tag{B.4}$$

If we apply $HYP \vdash \forall x \cdot P \longrightarrow HYP \vdash [x := E]P$ to B.2, we have:

$$[x := E](x = E \Longrightarrow P) \tag{B.5}$$

under the assumption B.4.

Applying substitution $[x := E](P \Longrightarrow Q) \longrightarrow [x := E]P \Longrightarrow [x := E]Q$ from Appendix B.2 to B.5 we obtain:

$$[x := E](x = E) \Longrightarrow (x := E)P \tag{B.6}$$

further leading to:

$$(E = E) \Longrightarrow (x := E)P \tag{B.7}$$

This leads to B.3 after applying Modus Ponens to B.7 under hypothesis B.2, since $E = E$ is true under any assumption. **End of Proof**

Lemma 2 $[x := E]P \Longrightarrow \forall x \cdot (x = E \Longrightarrow P),$ if $x \backslash E$

Proof:
Let us assume:

$$[x := E]P \tag{B.8}$$

Then we have to prove:

$$\forall x \cdot (x = E \Longrightarrow P) \tag{B.9}$$

under the following assumption:

$$x \backslash E \tag{B.10}$$

If we apply:

$$\left\{ \begin{array}{l} x \backslash H \textit{ for each } H \textit{ of } HYP \\ HYP \vdash Q \end{array} \right. \longleftrightarrow HYP \vdash \forall x \cdot Q \tag{B.11}$$

to B.9 with $x \backslash E$ as H and $\forall x \cdot (x = E \implies P)$ as Q, we have a new goal:

$$x = E \implies P \tag{B.12}$$

Now we assume:

$$x = E \tag{B.13}$$

and we have to prove:

$$P \tag{B.14}$$

If we apply Leibnitz Law:

$$\left\{ \begin{array}{l} HYP \vdash E = F \\ HYP \vdash [x := E]Q \end{array} \right. \longleftrightarrow HYP \vdash [x := F]Q \tag{B.15}$$

to B.8 and B.12 with $x = E \iff E = x$, we have:

$$[x := x]P \tag{B.16}$$

which directly leads to B.14, thus proving also B.9. **End of Proof**

Put together, Lemma 1 and Lemma 2 prove One Point Rule (B.1).

B.4 Type Checking

Elementary rules for type-checking of simple predicates P and Q are as follows:

$$\left\{ \begin{array}{l} \texttt{ENV} \vdash \texttt{check}(P) \\ \texttt{ENV} \vdash \texttt{check}(Q) \end{array} \right. \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(P \wedge Q)$$

$$\left\{ \begin{array}{l} \texttt{ENV} \vdash \texttt{check}(P) \\ \texttt{ENV} \vdash \texttt{check}(Q) \end{array} \right. \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(P \implies Q)$$

$$\texttt{ENV} \vdash \texttt{check}(P) \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(\neg P)$$

The following rules show how to solve quantified predicates. Let x and y be variables, s and t be sets, P, Q and R be predicates:

$$\begin{cases} x \backslash s \\ x \backslash R, \textit{for each } R \textit{ in } \texttt{ENV} \\ \texttt{ENV}, x \in s \vdash \texttt{check}(P) \end{cases} \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(\forall x \cdot (x \in s \Longrightarrow P))$$

$$\texttt{ENV} \vdash \texttt{check}(\forall x \cdot (x \in s \Longrightarrow \forall y \cdot (y \in t \Longrightarrow P)))$$

$$\longleftrightarrow ENV \vdash \texttt{check}(\forall (x, y) \cdot (x, y \in s \times t \Longleftrightarrow P))$$

$$\texttt{ENV} \vdash \texttt{check}(\forall x \cdot (P \Longrightarrow Q \wedge R)) \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(\forall x \cdot (P \wedge Q \Longrightarrow R))$$

Finally, we show the rules for equality, membership and inclusion. Let E and F be expressions, s and t sets:

$$\texttt{ENV} \vdash \texttt{type}(E) \equiv \texttt{type}(F) \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(E = F)$$

$$\texttt{ENV} \vdash \texttt{type}(E) \equiv \texttt{super}(s) \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(E \in s)$$

$$\texttt{ENV} \vdash \texttt{super}(s) \equiv \texttt{super}(t) \longleftrightarrow \texttt{ENV} \vdash \texttt{check}(s \subseteq t)$$

Appendix C

Algorithm for mapping CDL to AMN

constants = ∅, *properties* = ∅, *complex* = ∅, *sets* = ∅
var = ∅, *initialization* = ∅, *invariant* = ∅, *operations* = ∅
preconditions = ∅, *postconditions* = ∅ , *formalParams* = ∅
machine.name = *contract.attribute[serviceName]*
constants = *constants* ∪ *contract.attributes[* \ *serviceName]*
constants = *constants* ∪ *organization.elm[name]*
constants = *constants* ∪ *organization.elm[description]*
while (organization.elm[classification].hasNext())

    constants = *constants* ∪ *organization.classification.elm[*]*

constants = *constants* ∪ *organization.primaryContact.elm[*]*
constants = *constants* ∪ *location.elm[*]*
complex = *complex* ∪ *types.elm[targetNamespace]*
while (types.elm[complexType].hasNext())

    complex = *complex* ∪ *types.complexType.sequence.elm.attribute[*]*

while (contract.elm[event].hasNext())

    constants = *constants* ∪ *event.elm[*]*

foreach (method IN contract.elm[method])

    foreach (clause[precondition,postcondition,invariant] IN method)

        var = *var* ∪ *clause.log.elms[*].names*
        putClause(clause.log.elms[*]
        while (clause.sec.elm[authent].hasNext())

            var = *var* ∪ *clause.sec.authent.elm[credRef].names*
            putclause(clause.sec.authent.elm[credRef])

```
            while (clause.sec.elm[author].hasNext())
                var = var ∪ clause.sec.author.elm[secRole].names
                putClause(clause.sec.authotization.elm[secRole])
            var = var ∪ clause.dependability.transactions.elm[*].names
            putClause(clause.dependability.transactions.elm[*].names)
            var = var ∪ clause.dependability.repl.elm[*].names
            putClause(clause.dependability.repl.elm[*])
            var = var ∪ clause.dependability.checkpoint.elm[*].names
            putClause(clause.dependability.checkpoint.elm[*])
            var = var ∪ clause.dependability.timeout.attribute[*].names
            putClause(clause.dependability.timeout.attribute[*])
            while (clause.exceptions.app.elm[exc].hasNext())
                var = var ∪ clause.exceptions.app.exc.elm[*].names
                putClause(clause.exceptions.app.exc.elm[*])
            while (clause.exceptions.system.elm[exc].hasNext())
                var = var ∪ clause.exceptions.system.exc.elm[*].names
                putClause(clause.exceptions.system.exc.elm[*])
            while (clause.elm[performance].hasNext())
                var = var ∪ clause.performance.elm[*].names
                putClause(clause.performance.elm[*])
            while (clause.param.elm[param].hasNext())
                var = var ∪ method.param.param.elm[name].names
            foreach (set in method.params.elm[set])
                set = set ∪ method.params.set.elm[*]
        foreach (assertion IN method.elm[assertion])
            assertions = assertions ∪ method.assertion.elm[*]
putClause(elem)
case(elem)
        precondition: precondition = precondition ∧ elem
        postcondition: postcondition = postcondition ∧ elem
        invariant: invariant = invariant ∧ elem
if (elem.elm[initialization])
        initialization = initialization ∪ elem.elm[initialization]
        if (elem.attribute[direction] = INOUT)
            formalParams = formalParams ∪ elem)
```

Appendix D

Composition and Verification Example

In this appendix a composition example corresponding to the producer-consumer design pattern is shown. A correct composition is demonstrated, and then it is explained how different problems are detected during correctness verification. The classic producer-consumer scenario is considered, where values are produced, stored in a persistent storage and subsequently consumed. This flow is modeled using the following services: `producer`, `queue` (persistent storage), `consumer` and `start` (controls loop exit condition). The composition is shown in Figure D.1.

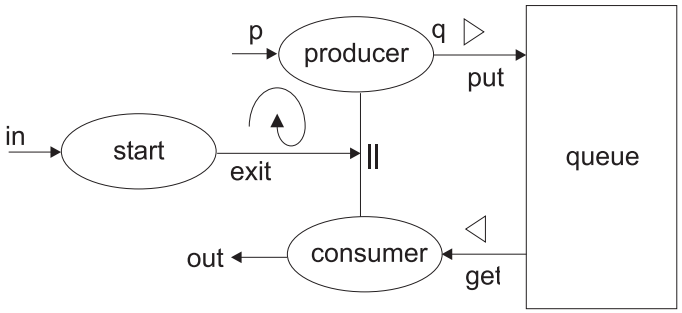

Figure D.1: Producer-consumer Composition

Suppose we want to model service composition describing part of a travel agency business flow using this example. The task is to create a composition that will describe receiving hotel reservation requests, their processing and sending responses to clients. Service `start` models a human employee who decides when the whole process will start and stop (e.g., every day from nine to five, day and night except Sunday, all year except during holidays,

etc.) by supplying parameter `in` that controls exit from a loop in which the remaining services are executing. Service `producer` models reservation receiving system. It can be a Web portal or a human employee that receives fax, letter, phone or direct request from the clients. From the perspective of composition modeling, we do not care how this is done, but only what is done. After a request is received it is stored in a persistent storage, which is modeled here with service `queue`. That means that this agency processes reservation requests by first-in first-out principle. If last-in first-out behavior is desired, a stack can be used. Service `consumer` models reservation confirmation system. It runs parallel to `producer` service and picks requests from the queue, checks availability and returns results to clients. Again, this can be done either automatically, or a human operator can manually fax or phone the hotel and confirm or cancel reservation.

The composition from Figure D.1 can be expressed in the following formula:

$$start \circlearrowleft_{(exit>0)} (producer \rhd queue)||(queue \rhd consumer)$$

The abstract machine describing `start` service is:

```
MACHINE start
CONSTANTS maxInt
PROPERTIES maxInt = 32768
SETS int = {0..maxInt}
VARIABLES in:IN, exit:OUT
INVARIANT exit ∈ int
OPERATIONS
exit <- start(in)
PRE in > 0 THEN
exit := in; exit := exit - 1; END
END
```

The abstract machine describing `queue` service is:

```
MACHINE queue (maxCapacity)
CONSTRAINTS maxCapacity ∈ ℕ
SETS QueueType, Authentication = {Kerberos, Oberon}, ℕ,
Authorization = {User, Administrator}
VARIABLES element:INOUT, queue, capacity:IN, status:OUT,
authentication:INOUT, authorization:INOUT
INVARIANT capacity ∈ [0, maxCapacity] ∧ queue ⊆ QueueType ∧
status ∈ N ∧ authentication ∈ Authentication ∧
authorization ∈ Authorization ∧ put ∈ QueueType → ℕ
∧ get ∈ {} → QueueType
INITIALIZATION capacity = maxCapacity, queue = ∅
OPERATIONS
status <- putInQueue(element) PRE element ∈ QueueType ∧ capacity > 0
```

```
∧ authentication = Kerberos ∧ authorization = Administrator THEN
queue := queue ∪ element ; capacity := capacity -1 ;
status := put(element) END
element <- getFromQueue() PRE capacity < maxCapacity THEN
element = get(); queue := queue \ element ;
capacity := capacity + 1 END
END
```

The machine producer is:

```
MACHINE producer
SETS QueueType, Authentication={Kerberos,Oberon},
Authorization={User,Administrator}
VARIABLES p:IN,q:OUT,authentication:INOUT,authorization:INOUT
INVARIANT q ∈ QueueType
OPERATIONS
q <- produce(p) PRE p ∈ QueueType
THEN q:=p; authorization:=Admin; authentication:=Kerberos END
END
```

The machine consumer is:

```
MACHINE consumer
SETS QueueType
VARIABLES r:IN,s:OUT
INVARIANT s ∈ QueueType
OPERATIONS
s <- consume(r) PRE r ∈ QueueType
s := r END
END
```

The composite machine agency is constructed as follows:

```
MACHINE agency(maxCapacity)
CONSTANTS maxInt
PROPERTIES maxInt=32768
CONSTRAINTS maxCpacity ∈ ℕ
SETS QueueType, Authentication={Kerberos,Oberon}, ℕ
Authorization={User,Administrator}, int= {0..maxInt}
VARIABLES p:IN,q,element:INOUT,queue,capacity,status,
authentication:INOUT,authorization:INPOUT,r,s:OUT
INVARIANT q ∈ QueueType ∧ capacity ∈ [0..maxCapacity]
∧ queue ⊆ QueueType ∧ status ∈ N ∧
authentication ∈ Authentication ∧ authorization ∈ Authorization
∧ s ∈ QueueType ∧ exit ∈ int ∧ put ∈ QueueType → ℕ
∧ get ∈ {} → QueueType
INITIALIZATION capacity = maxCapacity, queue = ∅
OPERATIONS
```

```
s<-agency(in,p)
PRE p ∈ QueueType ∧ element ∈ QueueType ∧ capacity > 0
∧ authentication = Kerberos ∧ authorization = Administrator
∧ capacity < maxCapacity ∧ r ∈ QueueType ∧ in > 0
THEN exit := in; exit:= exit-1;
WHILE (exit>0) DO
(q:=p; authorization:=Administrator; authentication:=Kerberos;
element:=q; queue:=queue ∪ element; capacity:=capacity-1;
status:=put(element))
||
(element:=get(); queue:=queue/element; capacity:=capacity+1;
r:=element; s:=r;)
exit:=exit-1; END
END
```

Correctness verification of the composed abstract machine agency begins by performing type checking. Machine given sets are distinct (QueueType, Authorization, Authentication and int), as well as constants maxInt and maxCapacity. Formal parameters and operations are not checked since there is only one operation agency and one formal machine parameter capacity. Given sets and constants are non-free in constraints, and state variables and machine formal parameters are non-free in properties. Then, environment for further type checking is formed:

$$\begin{gathered}
\texttt{given(maxCapacity,QueueType,Authentication,Authorization,int)},\\
\texttt{Kerberos} \in \texttt{Authentication}, \texttt{Oberon} \in \texttt{Authentication},\\
\texttt{User} \in \texttt{Authentication}, \texttt{Administrator} \in \texttt{Authentication} \vdash\\
\texttt{check}(\forall \texttt{maxCapacity} \cdot (\texttt{maxCapacity} \in \mathbb{N} \implies \forall(\texttt{maxInt}) \cdot \texttt{maxInt} = 32768 \implies\\
\forall(\texttt{p}, \texttt{q}, \texttt{element}, \texttt{queue}, \texttt{capacity}, \texttt{status}, \texttt{authentication}, \texttt{authorization}, \texttt{r}, \texttt{s}) \cdot\\
(\texttt{q} \in \texttt{QueueType} \wedge \texttt{capacity} \in (0..\texttt{maxCapacity}) \wedge \texttt{queue} \subseteq \texttt{QueueType} \wedge \texttt{status} \in N\\
\wedge \texttt{authentication} \in \texttt{Authentication} \wedge \texttt{authorization} \in \texttt{Authorization} \wedge\\
\texttt{s} \in \texttt{QueueType} \wedge \texttt{exit} \in \texttt{int} \implies O)
\end{gathered}$$

First machine formal parameter is checked together with constraints:

$$\begin{gathered}
\texttt{check}(\forall \texttt{maxCapacity} \cdot (\texttt{maxCapacity} \in \mathbb{N}))\\
\texttt{type}(\texttt{maxCapacity}) = \mathbb{N}
\end{gathered}$$

Then, constants and their properties are checked:

$$\texttt{check}(\forall(\texttt{maxInt} \cdot (\texttt{maxInt} = 32768))$$

$$\texttt{type}(\texttt{maxInt}) = \texttt{type}(32768) = \mathbb{N}$$

Now, variables and invariant are checked:

$$\texttt{check}(\forall(\texttt{p}, \texttt{q}, \texttt{element}, \texttt{queue}, \texttt{capacity}, \texttt{status}, \texttt{authentication}, \texttt{authorization}, \texttt{r}, \texttt{s}) \cdot$$

$$(\texttt{q} \in \texttt{QueueType} \land \texttt{capacity} \in (0..\texttt{maxCapacity}) \land \texttt{queue} \subseteq \texttt{QueueType} \land \texttt{status} \in N$$

$$\land \texttt{authentication} \in \texttt{Authentication} \land \texttt{authorization} \in \texttt{Authorization} \land$$

$$\texttt{s} \in \texttt{QueueType} \land \texttt{exit} \in \texttt{int}$$

$$\texttt{type}(\texttt{q}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{capacity}) = \texttt{super}(\{0..\texttt{maxCapacity}\}) = \mathbb{N}$$

$$\texttt{type}(\texttt{queue}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{status}) = \mathbb{N}$$

$$\texttt{type}(\texttt{authentication}) = \texttt{Authentication}$$

$$\texttt{type}(\texttt{authorization}) = \texttt{Authorization}$$

$$\texttt{type}(\texttt{s}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{exit}) = \texttt{int}$$

Finally, operation body is type checked, which by unwinding operation yields:

$$\texttt{type}(\texttt{p}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{element}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{r}) = \texttt{QueueType}$$

$$\texttt{type}(\texttt{in}) = \texttt{type}(\texttt{exit}) = \texttt{int}$$

The role of type-checking is to prepare machine for invariant preservation proofs. Type checking established the following:

- Composite machine is correctly formed because it obeys distinctiveness and non-freeness requirements.
- Machine defines all necessary sets in order to be self-contained, that is, to allow for typing of all variables.
- All pre-conditions are satisfied, as there are no conflicting requirements in the operation body. This will enable *assuming* pre-conditions when proving composite invariant in the next step.

Type checking can be violated by modifying the machine so that either of above requirements does not hold anymore. Suppose, for example, that we modify pre-condition such that instead of `PRE p ∈ QueueType` it becomes `PRE p ∈ OtherType`. Then type checking the invariant will give `type(q) = QueueType`, type checking pre-condition will give `type(p) = OtherType`, but type checking of the operation body will require that `type(q) = type(p)`, which will evaluate to false and thus signal a failed type checking.

Once type checking has been performed, we proceed to show that composite operation preserves composite invariant. We first prove that initialization establishes invariant, if constraints and properties are assumed:

$$\texttt{maxInt} = 32768 \wedge \texttt{maxCapacity} \in \mathbb{N} \Longrightarrow$$
$$[\texttt{capacity} = \texttt{maxCapacity} \wedge \texttt{queue} = \varnothing]$$
$$(\texttt{capacity} \in [0..\texttt{maxCapacity}] \wedge \texttt{queue} \subseteq \texttt{QueueType})$$

which evaluates to true expression:

$$\texttt{maxCapacity} \in [0..\texttt{maxCapacity}] \wedge \varnothing \subseteq \texttt{QueueType}$$

Now we show that composite operation preserves composite invariant, assuming constraints, properties, invariant and pre-conditions. For brevity, we show only that substitutions dealing with `capacity` preserve the invariant (in reality of course, all substitutions must be checked):

$$\texttt{maxCapacity} \in \mathbb{N} \wedge \texttt{capacity} \in [0..\texttt{maxCapacity}]$$
$$\wedge \texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity} \Longrightarrow$$
$$[\texttt{capacity} := \texttt{capacity} - 1 \parallel \texttt{capacity} := \texttt{capacity} + 1](\texttt{capacity} \in [0..\texttt{maxCapacity}])$$

This evaluates to true expression:

$$\texttt{capacity} - 1 \in [0..\texttt{maxCapacity}] || \texttt{capacity} + 1 \in [0..\texttt{maxCapacity}]$$

under the assumption:

$$\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}$$

Suppose that machine `queue` was defined such that pre-condition of `put` operation did not specify that capacity has to be positive. Then a part of the invariant preservation expression would be:

$$\texttt{capacity} \in [0..\texttt{maxCapacity}] \wedge \texttt{capacity} < \texttt{maxCapacity} \Longrightarrow$$
$$[\texttt{capacity} := \texttt{capacity} - 1](\texttt{capacity} \in [0..\texttt{maxCapacity}])$$

This operation will clearly be able to violate the invariant in case that producer tries to put element in the full queue (when $\texttt{capacity} = 0$). This problem could be solved by introducing scope guarded by an exception as demonstrated in the Chapter 7, but that would only facilitate handling of run-time errors. By requiring explicit invariant preservation proof this type of negative behavior is discovered at design time.

We finalize correctness verification by performing termination and feasibility checking. Again, we check only one pre-conditioned substitution:

$$\texttt{trm}(\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}|$$
$$\texttt{capacity} := \texttt{capacity} - 1||\texttt{capacity} := \texttt{capacity} + 1) =$$
$$\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity} \wedge$$
$$\texttt{trm}(\texttt{capacity} := \texttt{capacity} - 1||\texttt{capacity} := \texttt{capacity} + 1) =$$
$$\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity} \ \wedge$$
$$\texttt{trm}(\texttt{capacity} := \texttt{capacity} - 1) \wedge \texttt{trm}(\texttt{capacity} := \texttt{capacity} + 1)$$

Since $\texttt{trm}(\texttt{capacity} := \texttt{capacity} \pm 1) = \mathit{true}$, we get:

$$\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity} \wedge \mathit{true} \wedge \mathit{true} =$$
$$\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}$$

This result says that pre-conditioned substitution will terminate correctly if and only if its pre-conditions are satisfied. Up to now we have *assumed* that pre-conditions are satisfied and proved that machine is correct accordingly. That was good enough for design-time, since we have used generous specification model, where clients are expected to obey server pre-conditions. This is the place where we actually catch instantiated invalid pre-conditions. Note that the issue of concurrency in parallel substitutions is addressed by requiring that both pre-conditions hold before the parallel operation is executed. We do not know in advance which operation will take place first (increasing or decreasing capacity). By requiring that complex pre-condition $\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}$ holds, we prevent situations where producer executes first and tries to put an element in a full queue or where consumer executes first and tries to remove an element from an empty queue.

We consider loop termination now. Let us denote precondition with P and operation body with $A||B$; then the loop is:

```
PRE P THEN
exit:=in;
```

```
WHILE (exit > 0) DO
 I = A || B
 exit:=exit-1
END
```

What we have to prove is the following:

$$\forall \texttt{exit} \cdot (\texttt{A}||\texttt{B} \wedge \texttt{P} \Longrightarrow [n := \texttt{exit}][\texttt{exit} := \texttt{exit} - 1](\texttt{exit} < n)$$

Since $\texttt{exit} := \texttt{in}$ and $\texttt{in} > 0$, it follows:

$$\forall \texttt{exit} \cdot (\texttt{exit} > 0 \Longrightarrow \texttt{exit} - 1 < \texttt{exit})$$

Since this is obviously true, the loop will terminate for all values of `in`. However, in the case the loop has been defined with pre-condition $\alpha < 0$ and variant $\texttt{exit} := \texttt{exit} - \alpha$, termination expression would evaluate to false, since such loop would never terminate.

Finally, we perform feasibility check on the same pre-conditioned substitution:

$$\begin{gathered}
\texttt{fis}(\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity} \mid \\
\texttt{capacity} := \texttt{capacity} - 1 || \texttt{capacity} := \texttt{capacity} + 1) = \\
(\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}) \Longrightarrow \\
\texttt{fis}(\texttt{capacity} := \texttt{capacity} - 1 || \texttt{capacity} := \texttt{capacity} + 1) = \\
(\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}) \Longrightarrow \\
\texttt{fis}(\texttt{capacity} := \texttt{capacity} - 1) \wedge \texttt{fis}(\texttt{capacity} := \texttt{capacity} + 1)
\end{gathered}$$

Since $\texttt{fis}(\texttt{capacity} := \texttt{capacity} \pm 1) = \mathit{true}$, we get:

$$(\texttt{capacity} > 0 \wedge \texttt{capacity} < \texttt{maxCapacity}) \Longrightarrow \mathit{true}$$

The result means that pre-conditioned substitution is feasible even if its pre-conditions are not satisfied. This should not come as a surprise, since we know that pre-conditioned substitution will not establish anything (true or false) if pre-conditions do not hold. Since feasibility only checks whether a substitution is able to establish anything (in other words, is ambiguous or non-deterministic), it is clear that a substitution that will not take place (failed pre-conditioned substitution) is also feasible. On the other hand, assume that operation body consists of the following guarded substitution:

```
IF capacity > 0
 element := 'ABC'
output := element
```

Feasibility check would yield:

$$\texttt{fis}(\texttt{capacity} > 0 \Longrightarrow \texttt{element} := \texttt{'ABC'}) = \texttt{capacity} > 0 \wedge \texttt{fis}(\texttt{element} := \texttt{'ABC'}) =$$

$$\texttt{capacity} > 0 \wedge true$$

This substitution will be feasible only if $\texttt{capacity} > 0$, since otherwise subsequent assignment $\texttt{output}:=\texttt{element}$ is not defined, as we do not know the value of `element`. In case that conditional substitution is used instead of guarded, it would be feasible for all values of `capacity`.

Appendix E

Note on Domain Ontologies

It has been implicitly assumed in this work that partner services *understand* each other, that is, that they share a common knowledge and understanding of the world (e.g., parameter and method structure, name and meaning). This assumption has enabled us to develop a framework for verifiable and automatic composition without having to resolve the issue of semantic understanding on parameters and operations. The aforementioned assumption will be illustrated by two examples:

- In the Section 1.4 an example was presented where location of the mobile user is determined by composition of mobile location service and map service. In the course of composition, mobile location service provides a *location* object, map service accepts the same object and produces a map with the given location. It has been implicitly assumed, that both mobile location and map services share the common understanding on what the location is and how it is structured.

- In the Section 6.7, credit rating service communicates with two loan offer services, by exchanging user's credit rating as a parameter. Again, it has been assumed that all three services share the knowledge about the meaning of the parameter and the operation that is to be performed upon it.

The implicit assumption was that a common *domain ontology* exists, describing domain's objects and methods, which enables such shared understanding. While development of complete domain ontology language would be outside the scope of this work, there are elements in the proposed framework that actually describe parts of such ontology:

- `SETS` clause defines common sets and their structure (`SETS Type={Word,`

PDF,PS}). Since composite sets are generated by concatenation, all common types will be known.

- INVARIANT clause can determine semantic meaning of both service operations and parameter properties. For example, print ∈ Doc → Paper, describes a method *print* that transforms documents into paper (effectively describes printing a document). Also, fonts ∈ Doc → {Embedded,NotEmbedded} describes a property of a document called fonts, that can have value of being embedded or not.

- Complex type descriptions provide partner services with understanding on complex object structure. For example, description of location parameter required for the first example can be:

```
VARIABLES location, _x, _y, _z
COMPLEX location(_x, _y, _z)
SETS Float, Location
_x ∈ Float, _y ∈ Float, _z ∈ Float, location ∈ Location
```

- Cooperation graphs and classifications (Section 6.5), learning graphs (Section 6.6) and difference tables (Section 6.8) all represent domain ontologies, describing structure and properties of particular domains, objects and operations used within.

Another reason why systematic domain ontology language was not developed is that there are many current efforts and proposals in this area, which can all be used to provide a consistent framework for description of shared knowledge among services, such as Ontology Web Language for Services (OWL-S) [34], Resource Description Framework (RDF) [172], RosettaNet [148, 38], frame logic [76] and Web Service Modeling Language (WSML) [181].

For example, parts of the ontology required for the first example (user location) can be described using F-Logic in the following manner:

```
parameter[].
coordinate:parameter[x *=> float, y *=> float, z *=> float].
phoneNumber:parameter[countryCode *=> string, number *=> string].
person:parameter[name *=> string, phone *=> phoneNumber].
coordinateOf(person,coordinate).
coordinateOf(phoneNumber,coordinate).
coordinateOf(X,C) :- X[phoneNumber -> P], P:phoneNumber,
X:person, coordinateOf(P,C).
```

The equivalent description using WSML is:

```
concept phoneNumber
countryCode ofType xsd#string
number ofType xsd#string

concept coordinate
x of Type xsd#float
y of Type xsd#float
z of Type xsd#float

concept person
name typeOf xsd#string
phone typeOf xsd#string

relation coordinateOf(ofType phoneNumber, ofType coordinates)
relation coordinateOf(ofType person, ofType coordinates
axiom c definedBy coordinateOf(?x, ?c) impliedBy
?x member of person and ?p member of phoneNumber and
?c member of coordinates and coordinateOf(?p,?c)
and ?x[number hasValue ?p]
```

Both describe domain ontology where persons have coordinates, phones have coordinates, and when a person has a phone, person's coordinates can be determined by locating its mobile phone. Naturally, both languages are much more powerful as they allow for description of complex semantic relations that hold upon domain objects and operations.

Appendix F

CDL Database Schema

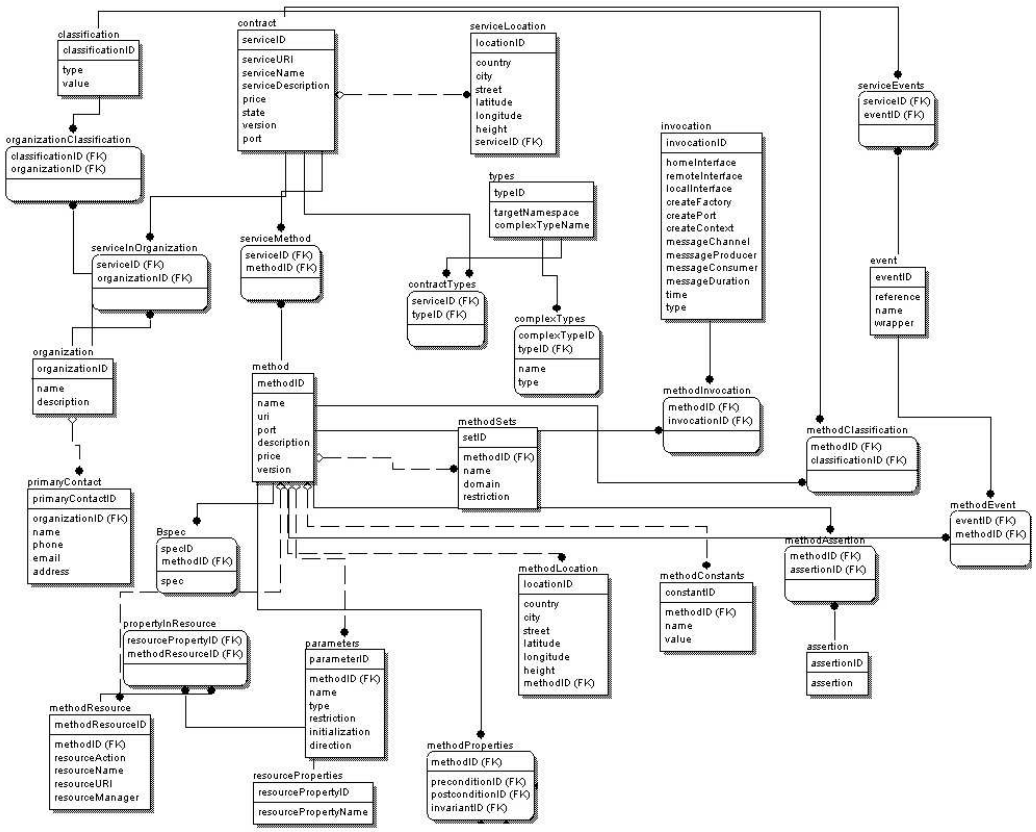

Figure F.1: CDL Database (part 1)

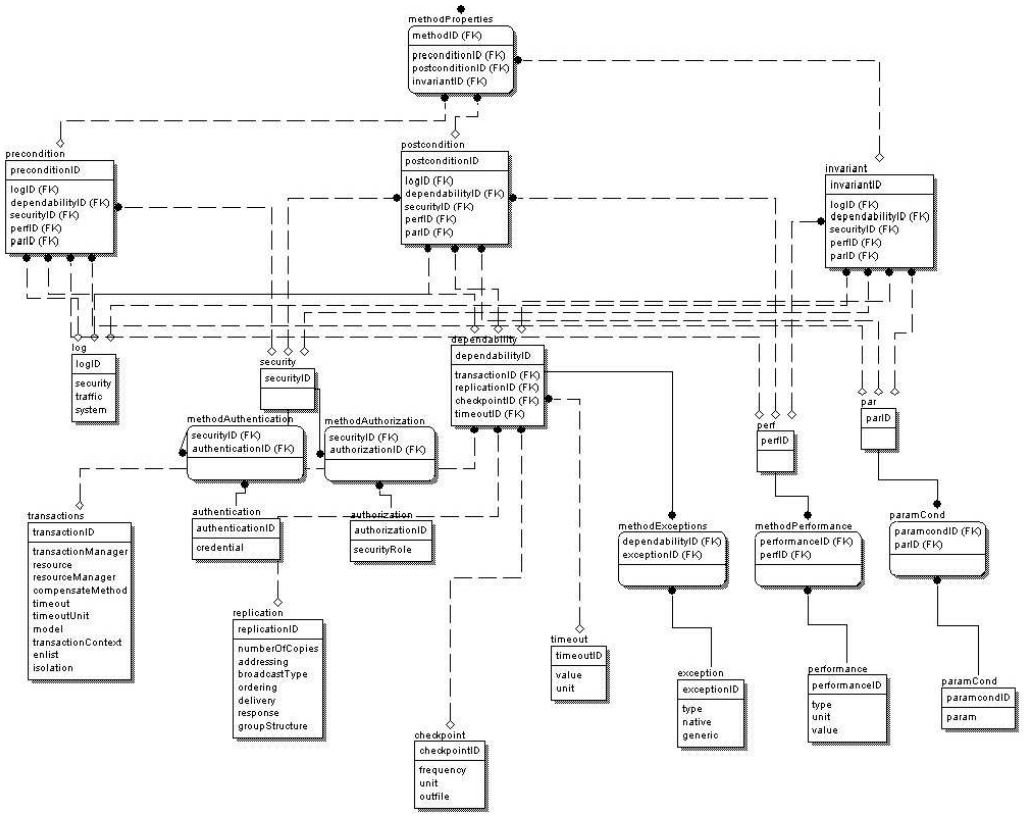

Figure F.2: CDL Database (part 2)

Appendix G

Client Interface

The process of working with the client interface consists of several distinct logical steps. A typical use-case is given:

- *Logging on to the system.* Client applications allows for two roles: `admin` and `client`. The only difference is that the `admin` role is allowed to perform administration of composition server and directory, invoking operations like adding, removing and changing published services, or setting parameters required for communication between composition server and underlying database and/or application servers hosting target services. The pure client does not have these options.

- *Search.* Regardless whether the composition server is used to invoke single or composite service, the search has to be performed first in order to find adequate service or composition partners. The upper-left pane from the Figure G.1 is used to perform basic search. Only the most frequently used fields are available in this pane. If the advances search is required (one that encompasses all properties defined in service contracts), `Advanced` buttons leads user to the dialog in which all relevant properties can be specified.

- *Browsing search results.* The search results are displayed in the lower-left pane. By selecting desired service and clicking on the `Details` button, user can display complete contract of the selected service. The search can then be either refined and performed again, or selected service can be moved to the composition pane (upper-right) by clicking `»Composition` button.

- *Composition, verification and execution.* The upper-right pane is used for composing selected services using graphical interface. Toolbar on

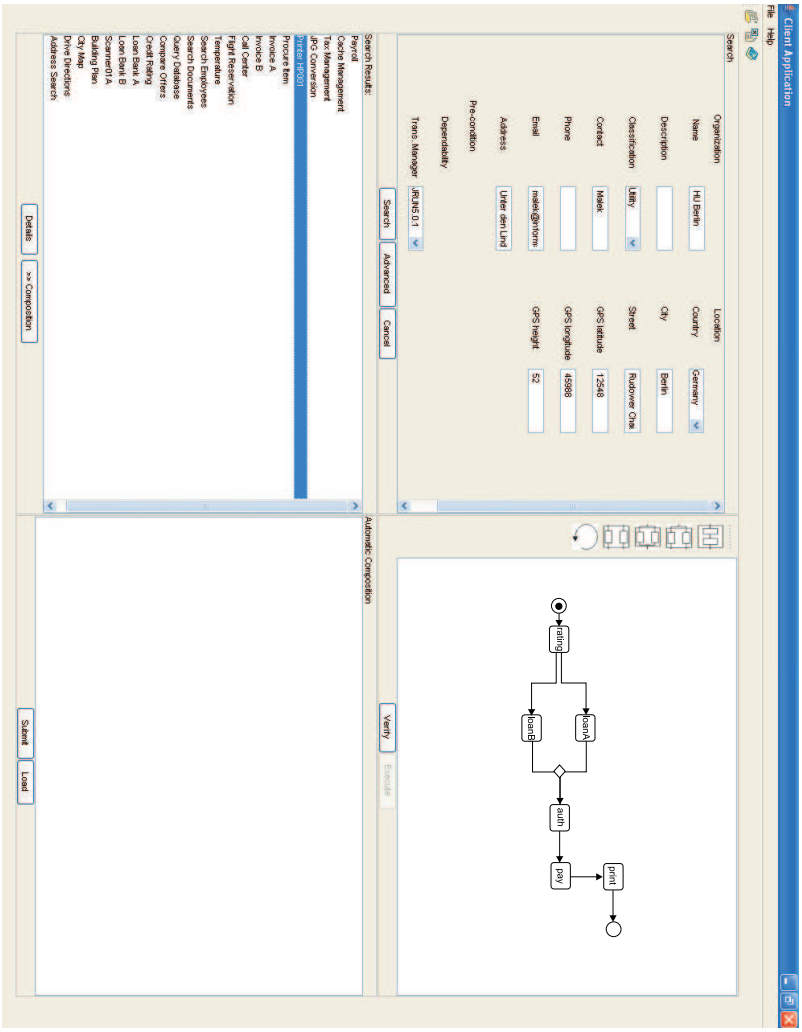

Figure G.1: Client Application

the right offers composition operators (sequence, choice, parallel with and without communication and loop), while selected services appear as boxes at the working panel. Composition is performed by selecting boxes representing services and connecting them with available operators. Composition can be undone, and operators can be also removed from a composition. Once complete composition is specified, clicking the `Verify` button invokes correctness verification which will display verification result in a separate window. If a composition is incorrect, diagnosis is shown, otherwise button `Execute` becomes active which allows for executing given composition. Execution results are also dis-

played in the separate window.

- *Automatic composition.* The lower-right pane offers the possibility of automatic service composition. It can be performed by typing target abstract machine in the provided panel, or by loading abstract machine description from a configuration file (button `Load`). After target machine is specified, it can be submitted to the composition server by clicking `Submit` button. If such composition exists, it will be displayed in the upper-right (composition) pane, ready for execution, otherwise message will be displayed that adequate composition could not be found with the suggestion to modify target abstract machine if possible.

Acknowledgements

First and foremost I would like to thank Miroslaw Malek, for his continued support throughout my work at the Humboldt University. My thanks also go to all the friends and colleagues. I am also grateful to my familly for their understanding, support, patience and love.

Selbständigkeitserklärung

Ich erkläre hiermit, daß

- ich die vorliegende Dissertationsschrift 'Contract-based Web Service Composition' selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universit"at zu Berlin bekannt ist.

Berlin, 07. März 2006
Nikola Milanovic